



# THE RAILS 3 WAY

Foreword by David H. Hansson creator of Ruby on Rails

OBIE FERNANDEZ

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 publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

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

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales (800) 382-3419 corpsales@pearsontechgroup.com

For sales outside the United States please contact: International Sales international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data Fernandez, Obie. The rails 3 way / Obie Fernandez. p. cm. Rev. ed. of: The Rails way / Obie Fernandez. 2008. Includes index. ISBN 0-321-60166-1 (pbk. : alk. paper) 1. Ruby on rails (Electronic resource) 2. Object-oriented programming (Computer science) 3. Ruby (Computer program language) 4. Web site development. 5. Application software–Development. I. Fernandez, Obie. Rails way. II. Title. QA76.64.F47 2010 005.1'17–dc22 2010038744

20100

Copyright © 2011 Pearson Education, Inc.

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

Pearson Education, Inc. Rights and Contracts Department 501 Boylston Street, Suite 900 Boston, MA 02116 Fax: (617) 671-3447

Parts of this book contain material excerpted from the Ruby and Rails source code and API documentation, Copyright © 2004–2011 by David Heinemeier Hansson under the MIT license. Chapter 18 contains material excerpted from the RSpec source code and API documentation, Copyright © 2005-2011 The RSpec Development Team.

The MIT License reads: Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT, OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OF OR OTHER DEALINGS IN THE SOFTWARE.

ISBN-13: 978-0-321-60166-7 ISBN-10: 0-321-60166-1

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan. First printing, December 2010

Editor-in-Chief Mark Taub

**Executive Acquisitions Editor** Debra Williams Cauley

Managing Editor John Fuller

Project Editor Elizabeth Ryan

Copy Editor Carol Loomis

Indexer Valerie Haynes Perry

**Proofreader** Erica Orloff

Publishing Coordinator Kim Boedigheimer

Cover Designer Chuti Prasertsith

**Compositor** Glyph International

## Contents

Foreword by David Heinemeier Hansson xxxiii Foreword by Yehuda Katz xxxv Introduction xxxvii Acknowledgments xliii About the Author xlv

#### Chapter 1 Rails Environments and Configuration 1

- 1.1 Bundler 2
  - 1.1.1 Gemfile 3
  - 1.1.2 Installing Gems 5
  - 1.1.3 Gem Locking 7
  - 1.1.4 Packaging Gems 7
- 1.2 Startup and Application Settings 8
  - 1.2.1 application.rb 8
  - 1.2.2 Initializers 11
  - 1.2.3 Additional Configuration 15
- 1.3 Development Mode 15
  - 1.3.1 Automatic Class Reloading 16
  - 1.3.2 Whiny Nils 18
  - 1.3.3 Error Reports 18
  - 1.3.4 Caching 18
  - 1.3.5 Raise Delivery Errors 19
- 1.4 Test Mode 19
- 1.5 Production Mode 20

- 1.5.1 Asset Hosts 22
- 1.5.2 Threaded Mode 22
- 1.6 Logging 23
  - 1.6.1 Rails Log Files 24
  - 1.6.2 Log File Analysis 26
- 1.7 Conclusion 29

#### Chapter 2 Routing 31

- 2.1 The Two Purposes of Routing 32
- 2.2 The routes.rb File 33
  - 2.2.1 Regular Routes 34
  - 2.2.2 URL Patterns 35
  - 2.2.3 Segment Keys 36
  - 2.2.4 Spotlight on the :id Field 38
  - 2.2.5 Optional Segment Keys 38
  - 2.2.6 Constraining Request Methods 38
  - 2.2.7 Redirect Routes 39
  - 2.2.8 The Format Segment 40
  - 2.2.9 Routes as Rack Endpoints 41
  - 2.2.10 Accept Header 42
  - 2.2.11 Segment Key Constraints 43
  - 2.2.12 The Root Route 44
- 2.3 Route Globbing 45
- 2.4 Named Routes 46
  - 2.4.1 Creating a Named Route 46
  - 2.4.2 name\_path vs. name\_url 47
  - 2.4.3 What to Name Your Routes 48
  - 2.4.4 Argument Sugar 49
  - 2.4.5 A Little More Sugar with Your Sugar? 50
- 2.5 Scoping Routing Rules 50
  - 2.5.1 Controller 51
  - 2.5.2 Path Prefix 51
  - 2.5.3 Name Prefix 52
  - 2.5.4 Namespaces 52
  - 2.5.5 Bundling Constraints 52
- 2.6 Listing Routes 53
- 2.7 Conclusion 54

#### Chapter 3 REST, Resources, and Rails 55

- 3.1 REST in a Rather Small Nutshell 55
- 3.2 Resources and Representations 56
- 3.3 REST in Rails 57
- 3.4 Routing and CRUD 58
  - 3.4.1 REST Resources and Rails 59
  - 3.4.2 From Named Routes to REST Support 59
  - 3.4.3 Reenter the HTTP Verb 60
- 3.5 The Standard RESTful Controller Actions 61
  - 3.5.1 Singular and Plural RESTful Routes 62
  - 3.5.2 The Special Pairs: new/create and edit/update 63
  - 3.5.3 The PUT and DELETE Cheat 64
  - 3.5.4 Limiting Routes Generated 64
- 3.6 Singular Resource Routes 64
- 3.7 Nested Resources 65
  - 3.7.1 RESTful Controller Mappings 66
  - 3.7.2 Considerations 67
  - 3.7.3 Deep Nesting? 67
  - 3.7.4 Shallow Routes 68
- 3.8 RESTful Route Customizations 69
  - 3.8.1 Extra Member Routes 70
  - 3.8.2 Extra Collection Routes 72
  - 3.8.3 Custom Action Names 72
  - 3.8.4 Mapping to a Different Controller 72
  - 3.8.5 Routes for New Resources 73
  - 3.8.6 Considerations for Extra Routes 73
- 3.9 Controller-Only Resources 74
- 3.10 Different Representations of Resources 76
  - 3.10.1 The respond\_to Method 76
  - 3.10.2 Formatted Named Routes 77
- 3.11 The RESTful Rails Action Set 78
  - 3.11.1 Index 78
  - 3.11.2 Show 80
  - 3.11.3 Destroy 80
  - 3.11.4 New and Create 81
  - 3.11.5 Edit and Update 82
- 3.12 Conclusion 83

#### Chapter 4 Working with Controllers 85

4.1	Rack	86	
	4.1.1	Configuring Your Middleware Stack 87	
4.2	Action	Dispatch: Where It All Begins 88	
	4.2.1	Request Handling 89	
	4.2.2	Getting Intimate with the Dispatcher 89	
4.3	Rende	r unto View 92	
	4.3.1	When in Doubt, Render 92	
	4.3.2	Explicit Rendering 93	
	4.3.3	Rendering Another Action's Template 93	
	4.3.4	Rendering a Different Template Altogether 94	
	4.3.5	Rendering a Partial Template 95	
	4.3.6	Rendering Inline Template Code 96	
	4.3.7	Rendering Text 96	
	4.3.8	Rendering Other Types of Structured Data 96	
	4.3.9	Rendering Nothing 97	
	4.3.10	Rendering Options 98	
4.4	Additional Layout Options 101		
4.5	Redire	cting 101	
	4.5.1	The redirect to Method 102	
4.6	Contro	oller/View Communication 104	
4.7	Filters	105	
	4.7.1	Filter Inheritance 106	
	4.7.2	Filter Types 107	
	4.7.3	Filter Chain Ordering 108	
	4.7.4	Around Filters 109	
	4.7.5	Filter Chain Skipping 110	
	4.7.6	Filter Conditions 110	
	4.7.7	Filter Chain Halting 111	
4.8	Verific	cation 111	
	4.8.1	Example Usage 111	
	4.8.2	Options 112	
4.9	Streaming 112		
	4.9.1	Viarender :text => proc 112	
	4.9.2	send data(data, options ={}) $113$	
	4.9.3	send file(path, options = $\{\}$ ) 114	
4.10	Concl	usion 117	

#### Chapter 5 Working with Active Record 119

5.1 The Basics 120 5.2 Macro-Style Methods 121 5.2.1 **Relationship Declarations** 121 5.2.2 Convention over Configuration 122 5.2.3 Setting Names Manually 122 5.2.4 Legacy Naming Schemes 122 5.3 **Defining** Attributes 123 Default Attribute Values 5.3.1 123 Serialized Attributes 5.3.2 125 5.4 CRUD: Creating, Reading, Updating, Deleting 127 Creating New Active Record Instances 5.4.1127 5.4.2 Reading Active Record Objects 128 5.4.3 Reading and Writing Attributes 128 5.4.4 Accessing and Manipulating Attributes Before They Are Typecast 131 5.4.5 Reloading 131 5.4.6 Cloning 131 5.4.7 Dynamic Attribute-Based Finders 132 5.4.8 Dynamic Scopes 133 5.4.9 Custom SQL Queries 133 5.4.10 The Query Cache 135 5.4.11 Updating 136 5.4.12 Updating by Condition 137 5.4.13 Updating a Particular Instance 138 5.4.14 Updating Specific Attributes 139 5.4.15 Convenience Updaters 139 5.4.16 Touching Records 139 5.4.17 Controlling Access to Attributes 140 5.4.18 Readonly Attributes 141 5.4.19 Deleting and Destroying 141 5.5 Database Locking 142 5.5.1 **Optimistic Locking** 143 5.5.2 Pessimistic Locking 145 5.5.3 Considerations 145 5.6 Where Clauses 146 5.6.1 where(\*conditions) 146

- 5.6.2 order(\*clauses) 5.6.3 limit (number) and offset (number) 149 5.6.4 select(\*clauses) 5.6.5 from(\*tables) 5.6.6 group(\*args) 150 5.6.7 having(\*clauses)
  - 5.6.8 includes(\*associations) 151
  - 5.6.9 joins 151
  - 5.6.10 readonly 152
  - 5.6.11 exists? 152
  - 5.6.12 arel table 152
- 5.7 Connections to Multiple Databases in Different Models 153

148

149

150

150

- 5.8 Using the Database Connection Directly 154
  - 5.8.1 The DatabaseStatements Module 154
  - 5.8.2 Other Connection Methods 156
- 5.9 Other Configuration Options 158
- 5.10 Conclusion 159

#### Chapter 6 Active Record Migrations 161

- 6.1 Creating Migrations 161
  - 6.1.1 Sequencing Migrations 162
  - 6.1.2 Irreversible Migrations 162
  - 6.1.3 create\_table(name, options, & block) 164
  - 6.1.4 change\_table(table\_name, & block) 165
  - 6.1.5 API Reference 165
  - 6.1.6 Defining Columns 167
  - Command-line Column Declarations 6.1.7 172
- 6.2 Data Migration 173
  - 6.2.1 Using SQL 173
  - 6.2.2 Migration Models 174
- 6.3 schema.rb 174
- 6.4 Database Seeding 175
- 6.5 Database-Related Rake Tasks 176
- 6.6 Conclusion 179

#### Chapter 7 Active Record Associations 181

- 7.1 The Association Hierarchy 181
- 7.2 **One-to-Many Relationships** 183

- 7.2.1 Adding Associated Objects to a Collection 185
- 7.2.2 Association Collection Methods 186
- 7.3 The belongs\_to Association 191
  - 7.3.1 Reloading the Association 192
  - 7.3.2 Building and Creating Related Objects via the Association 192
  - 7.3.3 belongs\_to Options 193
- 7.4 The has\_many Association 200
  - 7.4.1 has\_many Options 200
- 7.5 Many-to-Many Relationships 209
  - 7.5.1 has\_and\_belongs\_to\_many 209
  - 7.5.2 has\_many :through 215
  - 7.5.3 has\_many :through Options 220
- 7.6 One-to-One Relationships 223
  - 7.6.1 has\_one 223
- 7.7 Working with Unsaved Objects and Associations 226
  - 7.7.1 One-to-One Associations 226
  - 7.7.2 Collections 226
  - 7.7.3 Deletion 227
- 7.8 Association Extensions 227
- 7.9 The AssociationProxy Class 229
- 7.10 Conclusion 230

#### Chapter 8 Validations 231

8.1	Finding	Errors 231
8.2	The Sin	nple Declarative Validations 232
	8.2.1	validates_acceptance_of 232
	8.2.2	validates_associated 233
	8.2.3	validates_confirmation_of 233
	8.2.4	validates_each 234
	8.2.5	validates_format_of 235
	8.2.6	validates_inclusion_of
		and validates_exclusion_of 236
	8.2.7	validates_length_of 236
	8.2.8	validates_numericality_of 237
	8.2.9	validates_presence_of 238
	8.2.10	validates_uniqueness_of 239
	8.2.11	validates_with 241
	8.2.12	RecordInvalid 242

- 8.3 Common Validation Options 242
  - 8.3.1 :allow\_blank and :allow\_nil 242
  - 8.3.2 :if and :unless 242
  - 8.3.3 :message 242
  - 8.3.4 :on 243
- 8.4 Conditional Validation 243
  - 8.4.1 Usage and Considerations 244
  - 8.4.2 Validation Contexts 245
- 8.5 Short-form Validation 245
- 8.6 Custom Validation Techniques 246
  - 8.6.1 Add Custom Validation Macros to Your Application 247
  - 8.6.2 Create a Custom Validator Class 248
  - 8.6.3 Add a validate Method to Your Model 248
- 8.7 Skipping Validations 249
- 8.8 Working with the Errors Hash 249
  - 8.8.1 Checking for Errors 250
- 8.9 Testing Validations with Shoulda 250
- 8.10 Conclusion 250

#### Chapter 9 Advanced Active Record 251

- 9.1 Scopes 251
  - 9.1.1 Scope Parameters 252
  - 9.1.2 Chaining Scopes 252
  - 9.1.3 Scopes and has\_many 252
  - 9.1.4 Scopes and Joins 253
  - 9.1.5 Scope Combinations 253
  - 9.1.6 Default Scopes 254
  - 9.1.7 Using Scopes for CRUD 255

#### 9.2 Callbacks 256

- 9.2.1 Callback Registration 256
- 9.2.2 One-Liners 257
- 9.2.3 Protected or Private 257
- 9.2.4 Matched before/after Callbacks 258
- 9.2.5 Halting Execution 259
- 9.2.6 Callback Usages 259
- 9.2.7 Special Callbacks: after\_initialize and after\_find 262
- 9.2.8 Callback Classes 263

9.3 Calculation Methods 265 9.3.1 average(column name, \*options) 267 9.3.2 count(column name, \*options) 267 9.3.3 maximum(column name, \*options) 267 9.3.4 minimum(column name, \*options) 267 9.3.5 sum(column name, \*options) 267 9.4 Observers 268 9.4.1 Naming Conventions 268 9.4.2 Registration of Observers 269 9.4.3 Timing 269 9.5 Single-Table Inheritance (STI) 269 9.5.1 Mapping Inheritance to the Database 271 9.5.2 STI Considerations 273 9.5.3 STI and Associations 2749.6 Abstract Base Model Classes 276 Polymorphic has\_many Relationships 9.7 277 9.7.1 In the Case of Models with Comments 278 9.8 Foreign-key Constraints 281 9.9 Using Value Objects 281 9.9.1 Immutability 283 9.9.2 Custom Constructors and Converters 283 9.9.3 Finding Records by a Value Object 284 9.10 Modules for Reusing Common Behavior 285 9.10.1 A Review of Class Scope and Contexts 287 The included Callback 288 9.10.2 9.11 Modifying Active Record Classes at Runtime 289 9.11.1 Considerations 290 Ruby and Domain-Specific Languages 9.11.2 291 9.12 Conclusion 292 Chapter 10 Action View 293 10.1 Layouts and Templates 294 10.1.1 Template Filename Conventions 294 10.1.2 Layouts 294 10.1.3 Yielding Content 295 10.1.4 Conditional Output 296 10.1.5 Decent Exposure 297

- 10.1.6 Standard Instance Variables 298
- 10.1.7 Displaying flash Messages 300
- 10.1.8 flash.now 301

#### 10.2 Partials 302

- 10.2.1 Simple Use Cases 302
- 10.2.2 Reuse of Partials 303
- 10.2.3 Shared Partials 304
- 10.2.4 Passing Variables to Partials 305
- 10.2.5 Rendering Collections 306
- 10.2.6 Logging 308
- 10.3 Conclusion 308

#### Chapter 11 All About Helpers 309

- 11.1 ActiveModelHelper 309
  - 11.1.1 Reporting Validation Errors 310
  - 11.1.2 Automatic Form Creation 313
  - 11.1.3 Customizing the Way Validation Errors Are Highlighted 315
- 11.2 AssetTagHelper 316
  - 11.2.1 Head Helpers 316
  - 11.2.2 Asset Helpers 319
  - 11.2.3 Using Asset Hosts 321
  - 11.2.4 Using Asset Timestamps 323
  - 11.2.5 For Plugins Only 324
- 11.3 AtomFeedHelper 324
- 11.4 CacheHelper 326
- 11.5 CaptureHelper 326
- 11.6 DateHelper 328
  - 11.6.1 The Date and Time Selection Helpers 328
  - 11.6.2 The Individual Date and Time Select Helpers 329
  - 11.6.3 Common Options for Date Selection Helpers 332
  - 11.6.4 distance\_in\_time Methods with Complex Descriptive Names 332
- 11.7 DebugHelper 333
- 11.8 FormHelper 333
  - 11.8.1 Creating Forms for Models 334
  - 11.8.2 How Form Helpers Get Their Values 342
  - 11.8.3 Integrating Additional Objects in One Form 343

- 11.8.4 Customized Form Builders 347
- 11.8.5 Form Inputs 348
- 11.9 FormOptionsHelper 350
  - 11.9.1 Select Helpers 350
  - 11.9.2 Option Helpers 351
- 11.10 FormTagHelper 355
- 11.11 JavaScriptHelper 358
- 11.12 NumberHelper 359
- 11.13 PrototypeHelper 361
- 11.14 RawOutputHelper 361
- 11.15 RecordIdentificationHelper 362
- 11.16 RecordTagHelper 363
- 11.17 SanitizeHelper 364
- 11.18 TagHelper 366
- 11.19 TextHelper 367
- 11.20 TranslationHelper and the I18n API 372
  - 11.20.1 Localized Views 373
  - 11.20.2 TranslationHelper Methods 374
  - 11.20.3 I18n Setup 374
  - 11.20.4 Setting and Passing the Locale 375
  - 11.20.5 Setting Locale from Client Supplied Information 379
  - 11.20.6 Internationalizing Your Application 380
  - 11.20.7 Organization of Locale Files 382
  - 11.20.8 Looking up Translations 383
  - 11.20.9 How to Store Your Custom Translations 386
  - 11.20.10 Overview of Other Built-In Methods that Provide I18n Support 388
  - 11.20.11 Exception Handling 391
- 11.21 UrlHelper 391
- 11.22 Writing Your Own View Helpers 398
  - 11.22.1 Small Optimizations: The Title Helper 398
  - 11.22.2 Encapsulating View Logic: The photo\_for Helper 399
  - 11.22.3 Smart View: The breadcrumbs Helper 400
- 11.23 Wrapping and Generalizing Partials 401
  - 11.23.1 A tiles Helper 401
  - 11.23.2 Generalizing Partials 404
- 11.24 Conclusion 407

#### Chapter 12 Ajax on Rails 409

```
12.0.1
             Changes in Rails 3 410
      12.0.2
             Firebug 410
12.1 Unobtrusive JavaScript 411
     12.1.1 UJS Usage 411
    Writing JavaScript in Ruby with RJS 412
12.2
     12.2.1 RJS Templates 414
     12.2.2 <<(javascript) 415
      12.2.3 [](id) 415
     12.2.4 alert(message) 416
      12.2.5 call(function, *arguments, & block)
                                                416
      12.2.6 \text{ delay(seconds} = 1) \dots 416
      12.2.7 draggable(id, options = \{\}) 416
      12.2.8 drop_receiving(id, options = \{\}) 417
     12.2.9 hide(*ids) 417
      12.2.10 insert html (position, id, *options for render)
                                                                       417
      12.2.11 literal(code) 417
     12.2.12 redirect_to(location)
                                  418
      12.2.13 remove(*ids) 418
      12.2.14 replace(id, *options_for_render) 418
      12.2.15 replace_html(id, *options_for_render) 418
      12.2.16 select(pattern) 418
     12.2.17 show(*ids) 418
     12.2.18 sortable(id, options = \{\}) 418
      12.2.19 toggle(*ids) 419
      12.2.20 visual_effect(name, id = nil, options = \{\}) 419
     Ajax and ISON 419
12.3
     12.3.1 Ajax link_to 419
12.4 Ajax and HTML 421
12.5 Ajax and JavaScript 423
12.6
     Conclusion 424
```

#### Chapter 13 Session Management 425

- 13.1 What to Store in the Session 426
  - 13.1.1 The Current User 426
  - 13.1.2 Session Use Guidelines 426

- 13.2 Session Options 427
- 13.3 Storage Mechanisms 427
  - 13.3.1 Active Record Session Store 427
  - 13.3.2 Memcache Session Storage 428
  - 13.3.3 The Controversial CookieStore 429
  - 13.3.4 Cleaning Up Old Sessions 430
- 13.4 Cookies 431 13.4.1 Reading and Writing Cookies 431
- 13.5 Conclusion 432

#### Chapter 14 Authentication 433

- 14.1 Authlogic 434
  - 14.1.1 Getting Started 434
  - 14.1.2 Creating the Models 434
  - 14.1.3 Setting Up the Controllers 435
  - 14.1.4 Controller, Limiting Access to Actions 436
  - 14.1.5 Configuration 437
  - 14.1.6 Summary 439
- 14.2 Devise 439
  - 14.2.1 Getting Started 439
  - 14.2.2 Modules 439
  - 14.2.3 Models 440
  - 14.2.4 Controllers 441
  - 14.2.5 Devise, Views 442
  - 14.2.6 Configuration 442
  - 14.2.7 Extensions 443
  - 14.2.8 Summary 443
- 14.3 Conclusion 443

#### Chapter 15 XML and Active Resource 445

15.1 The to\_xml Method 445

- 15.1.1 Customizing to\_xml Output 446
- 15.1.2 Associations and to\_xml 448
- 15.1.3 Advanced to\_xml Usage 451
- 15.1.4 Dynamic Runtime Attributes 452
- 15.1.5 Overriding to\_xml 453

- 15.2 The XML Builder 454
- 15.3 Parsing XML 456 15.3.1 Turning XML into Hashes 456
  - 15.3.2 Typecasting 457
- 15.4 Active Resource 457
  - 15.4.1 List 458
  - 15.4.2 Show 459
  - 15.4.3 Create 460
  - 15.4.4 Update 462
  - 15.4.5 Delete 462
  - 15.4.6 Headers 462
  - 15.4.7 Customizing URLs 463
  - 15.4.8 Hash Forms 464
- 15.5 Active Resource Authentication 465
  - 15.5.1 HTTP Basic Authentication 465
  - 15.5.2 HTTP Digest Authentication 466
  - 15.5.3 Certificate Authentication 466
  - 15.5.4 Proxy Server Authentication 466
  - 15.5.5 Authentication in the Web Service Controller 467
- 15.6 Conclusion 469

#### Chapter 16 Action Mailer 471

- 16.1 Setup 471
- 16.2 Mailer Models 472
  - 16.2.1 Preparing Outbound Email Messages 472
  - 16.2.2 HTML Email Messages 474
  - 16.2.3 Multipart Messages 475
  - 16.2.4 Attachments 475
  - 16.2.5 Generating URLs 476
  - 16.2.6 Mailer Layouts 476
  - 16.2.7 Sending an Email 477
- 16.3 Receiving Emails 477
  - 16.3.1 Handling Incoming Attachments 478
- 16.4 Server Configuration 479
- 16.5 Testing Email Content 479
- 16.6 Conclusion 481

#### xxii

#### Chapter 17 Caching and Performance 483

17.1	View Caching 483	
	17.1.1 Caching in Development Mode? 484	
	17.1.2 Page Caching 484	
	17.1.3 Action Caching 484	
	17.1.4 Fragment Caching 486	
	17.1.5 Expiration of Cached Content 488	
	17.1.6 Automatic Cache Expiry with Sweepers 490	
	17.1.7 Cache Logging 492	
	17.1.8 Action Cache Plugin 492	
	17.1.9 Cache Storage 493	
17.2	General Caching 495	
	17.2.1 Eliminating Extra Database Lookups 495	
	17.2.2 Initializing New Caches 496	
	17.2.3 fetch Options 496	
17.3	Control Web Caching 497	
	$17.3.1 \text{ expires_in(seconds, options = }) 498$	
	17.3.2 expires_now 498	
17.4	ETags 498	
	17.4.1 fresh_when(options) 499	
	17.4.2 stale?(options) 499	
17.5	Conclusion 500	

504 506

#### Chapter 18 RSpec 501

-		•
18.1	Introdu	ction 501
18.2	Basic Sy	ntax and API 504
	18.2.1	describe and context 504
	18.2.2	<pre>let(:name) (expression)</pre>
	18.2.3	<pre>let!(:name) (expression)</pre>
	18.2.4	before and after 506
	18.2.5	it 507
	18.2.6	specify 507
	18.2.7	expect 508
	18.2.8	pending 509
	18.2.9	should and should_not $510$
	18.2.10	Implicit Subject 511

18.2.11 Explicit Subject 511 18.2.12 its 512 18.3 Predicate Matchers 513 18.4 Custom Expectation Matchers 514 18.4.1 Custom Matcher DSL 516 18.4.2 Fluent Chaining 516 Shared Behaviors 517 18.5 18.6 RSpec's Mocks and Stubs 517 Running Specs 520 18.7 18.8 RSpec Rails Gem 521 18.8.1 Installation 521 18.8.2 Model Specs 524 18.8.3 Mocked and Stubbed Models 18.8.4 Controller Specs 526 18.8.5 View Specs 529 18.8.6 Helper Specs 531 18.9 RSpec Tools 531 18.9.1 RSpactor 531 18.9.2 watchr 532 18.9.3 Spork 532 18.9.4 Specjour 532 18.9.5 RCov 532

526

- 18.9.6 Heckle 532
- 18.10 Conclusion 533

#### Chapter 19 Extending Rails with Plugins 535

- 19.1 The Plugin System 536
  - 19.1.1 Plugins as RubyGems 536
  - 19.1.2 The Plugin Script 536
- 19.2 Writing Your Own Plugins 537
  - 19.2.1 The init.rb Hook 538
  - 19.2.2 The lib Directory 539
  - 19.2.3 Extending Rails Classes 540
  - 19.2.4 The README and MIT-LICENSE File 541
  - 19.2.5 The install.rb and uninstall.rb Files 542
  - 19.2.6 Custom Rake Tasks 543

19.2.7 The Plugin's Rakefile 544
19.2.8 Including Assets With Your Plugin 545
19.2.9 Testing Plugins 545
19.2.10 Railties 546
19.3 Conclusion 547

#### Chapter 20 Background Processing 549

- 20.1 Delayed Job 550
  - 20.1.1 Getting Started 550
  - 20.1.2 Creating Jobs 551
  - 20.1.3 Running 552
  - 20.1.4 Summary 552
- 20.2 Resque 553
  - 20.2.1 Getting Started 553
  - 20.2.2 Creating Jobs 554
  - 20.2.3 Hooks 554
  - 20.2.4 Plugins 555
  - 20.2.5 Running 556
  - 20.2.6 Monitoring 556
  - 20.2.7 Summary 557
- 20.3 Rails Runner 557
  - 20.3.1 Getting Started 558
  - 20.3.2 Usage Notes 558
  - 20.3.3 Considerations 559
  - 20.3.4 Summary 559
- 20.4 Conclusion 559

#### Appendix A Active Model API Reference 561

```
A.1 AttributeMethods 561

A.1.1 active_model/attribute_methods.rb 562

A.2 Callbacks 563

A.2.1 active_model/callbacks.rb 563

A.3 Conversion 563

A.3.1 active_model/conversion.rb 563

A.4 Dirty 564

A.4.1 active_model/dirty.rb 565
```

A.5	Errors 565		
	A.5.1 active_model/errors.rb 566		
A.6	Lint::Tests 567		
A.7	MassAssignmentSecurity 567		
	A.7.1 active_model/mass_assignment_security.rb 567		
A.8	Name 568		
	A.8.1 active_model/naming.rb 569		
A.9	Naming 569		
	A.9.1 active_model/naming.rb 569		
A.10	Observer 569		
	A.10.1 active_model/observing.rb 570		
A.11	Observing 570		
	A.11.1 active_model/observing.rb 571		
A.12	Serialization 571		
	A.12.1 active_model/serialization.rb 571		
A.13	Serializers::JSON 572		
	A.13.1 active_model/serializers/json.rb 572		
A.14	Serializers::Xml 572		
	A.14.1 active_model/serializers/xml.rb 573		
A.15	Translation 573		
	A.15.1 active_model/translation.rb 573		
A.16	Validations 574		
	A.16.1 active_model/validations.rb 574		
A.17	Validator 578		
	A.17.1 active_model/validator.rb 578		

### Appendix B Active Support API Reference 579

B.1	Array	579
	B.1.1	active_support/core_ext/array/access 579
	B.1.2	active_support/core_ext/array/conversions 580
	B.1.3	active_support/core_ext/array/
		extract_options 582
	B.1.4	active_support/core_ext/array/grouping 583
	B.1.5	$\verb+active\_support/core\_ext/array/random\_access-584$
	B.1.6	active_support/core_ext/array/uniq_by 584
	B.1.7	active_support/core_ext/array/wrap 584

#### Contents

B.1.8 active support/core\_ext/object/blank 585 B.1.9 active support/core ext/object/to param 585 B.2 ActiveSupport::BacktraceCleaner 585 B.2.1 active support/backtrace cleaner 585 B.3 ActiveSupport::Base64 586 B.3.1 active support/base64 586 B.4 ActiveSupport::BasicObject 586 B.4.1 active support/basic object 586 B.5 ActiveSupport::Benchmarkable 587 B.5.1 active support/benchmarkable 587 B.6 BigDecimal 588 B.6.1 active\_support/core\_ext/big\_decimal/ conversions 588 B.6.2 active\_support/json/encoding 588 B.7 ActiveSupport::BufferedLogger 588 B.7.1 active support/buffered logger 589 B.8 ActiveSupport::Cache::Store 590 B.9 ActiveSupport::Callbacks 595 B.9.1 active support/callbacks 596 B.10 Class 598 B.10.1 active support/core\_ext/class/attribute 598 B.10.2 active support/core ext/class/ attribute accessors 599 B.10.3 active\_support/core\_ext/class/ attribute accessors 600 B.10.4 active support/core ext/class/ delegating attributes 600 B.10.5 active support/core ext/class/ inheritable attributes 600 B.10.6 active\_support/core\_ext/class/subclasses 601 B.11 ActiveSupport::Concern 602 B.11.1 active support/concern 602 B.12 ActiveSupport::Configurable 603 B.12.1 active\_support/configurable 603 B.13 Date 603 B.13.1 active support/core ext/date/acts like 603 B.13.2 active support/core ext/date/calculations 603

	B.13.3 active_support/core_ext/date/conversions 607
	$B.13.4$ active_support/core_ext/date/freeze $608$
	B.13.5 active_support/json/encoding 609
B.14	DateTime 609
	B.14.1 active_support/core_ext/date_time/acts_like 609
	$B.14.2$ active_support/core_ext/date_time/calculations $609$
	B.14.3 active_support/core_ext/date_time/conversions 611
	$B.14.4$ active_support/core_ext/date_time/zones $612$
	B.14.5 active_support/json/encoding 613
B.15	ActiveSupport::Dependencies 613
	B.15.1 active_support/dependencies/autoload $614$
B.16	ActiveSupport::Deprecation $617$
B.17	ActiveSupport::Duration 617
	B.17.1 active_support/duration 617
B.18	Enumerable 619
	B.18.1 active_support/core_ext/enumerable 619
	B.18.2 active_support/json/encoding $620$
B.19	ERB::Util 620
	B.19.1 active_support/core_ext/string/output_safety $620$
B.20	FalseClass 621
	B.20.1 active_support/core_ext/object/blank 621
	B.20.2 active_support/json/encoding 621
B.21	File 621
	B.21.1 active_support/core_ext/file/atomic 621
	B.21.2 active_support/core_ext/file/path 622
B.22	Float 622
	B.22.1 active_support/core_ext/float/rounding 622
B.23	Hash 622
	B.23.1 active_support/core_ext/hash/conversions 622
	B.23.2 active_support/core_ext/hash/deep_merge 623
	B.23.3 active_support/core_ext/hash/diff 624
	B.23.4 active_support/core_ext/hash/except 624
	B.23.5 active_support/core_ext/hash/
	indifferent_access 624
	B.23.6 active_support/core_ext/hash/keys 625

B.23.7 active\_support/core\_ext/hash/reverse\_merge 626

B.23.8 active support/core ext/hash/slice 626 B.23.9 active support/core ext/object/to param 627 B.23.10 active support/core ext/object/to guery 627 B.23.11 active support/json/encoding 627 B.23.12 active support/core ext/object/blank 627 B.24 HashWithIndifferentAccess 627 B.24.1 active support/hash with indifferent access 627 B.25 ActiveSupport::Inflector::Inflections 628 B.25.1 active support/inflector/inflections 629 B.25.2 active support/inflector/transliteration 631 B.26 Integer 632 B.26.1 active\_support/core\_ext/integer/inflections 633 B.26.2 active support/core ext/integer/multiple 633 B.27 ActiveSupport:: JSON 633 B.27.1 active support/json/decoding 633 B.27.2 active\_support/json/encoding 634 B.28 Kernel 634 B.28.1 active support/core ext/kernel/agnostics 634 B.28.2 active\_support/core\_ext/kernel/debugger 634 B.28.3 active support/core ext/kernel/reporting 634 B.28.4 active support/core ext/kernel/requires 635 B.28.5 active support/core ext/kernel/ singleton class 635 B.29 Logger 635 B.29.1 active support/core ext/logger 636 B.30 ActiveSupport::MessageEncryptor 636 B.30.1 active support/message encryptor 637 B.31 ActiveSupport::MessageVerifier 637 B.31.1 active support/message verifier 637 B.32 Module 638 B.32.1 active support/core ext/module/aliasing 638 B.32.2 active support/core ext/module/anonymous 639 B.32.3 active support/core\_ext/module/ attr accessor with default 640 B.32.4 active\_support/core\_ext/module/attr\_internal 640 B.32.5 active\_support/core\_ext/module/ attribute accessors 640

	B.32.6	active_support/core_ext/module/delegation 641
	B.32.7	active_support/core_ext/module/introspection 643
	B.32.8	active_support/core_ext/module/
		synchronization 644
	B.32.9	active_support/dependencies 644
B.33	Active	eSupport::Multibyte::Chars 645
	B.33.1	active_support/multibyte/chars 645
	B.33.2	active_support/multibyte/unicode 646
	B.33.3	active_support/multibyte/utils 647
B.34	NilCla	ass 648
	B.34.1	active_support/core_ext/object/blank 648
	B.34.2	active_support/json/encoding 648
	B.34.3	active_support/whiny_nil 648
B.35	ActiveSupport::Notifications 649	
B.36	Numeri	ic 650
	B.36.1	active_support/core_ext/object/blank 650
	B.36.2	active_support/json/encoding $650$
	B.36.3	active_support/numeric/bytes 650
	B.36.4	active_support/numeric/time 651
B.37	Object	z 653
	B.37.1	active_support/core_ext/object/acts_like 653
	B.37.2	active_support/core_ext/object/blank 653
	B.37.3	active_support/core_ext/object/duplicable 654
	B.37.4	active_support/core_ext/object/
		instance_variables 654
	B.37.5	active_support/core_ext/object/to_param 655
	B.37.6	active_support/core_ext/object/with_options 656
	B.37.7	active_support/dependencies 656
	B.37.8	active_support/json/encoding 657
B.38	Active	eSupport::OrderedHash 657
	B.38.1	active_support/ordered_hash 657
B.39	Active	eSupport::OrderedOptions 657
	B.39.1	active_support/ordered_options 657
B.40	Active	eSupport::Railtie 658
	B.40.1	active_support/railtie 658

B.41 Range 658

	$B.41.1$ active_support/core_ext/range/blockless_step $658$
	B.41.2 active_support/core_ext/range/conversions 659
	B.41.3 active_support/core_ext/range/include_range 659
	B.41.4 active_support/core_ext/range/include_range 659
B.42	Regexp 660
	$B.42.1$ active_support/core_ext/enumerable $660$
	B.42.2 active_support/json/encoding $660$
B.43	ActiveSupport::Rescuable 660
	B.43.1 active_support/rescuable 660
B.44	ActiveSupport::SecureRandom 661
	B.44.1 active_support/secure_random 661
B.45	String 662
	B.45.1 active_support/json/encoding 662
	B.45.2 active_support/core_ext/object/blank $662$
	B.45.3 active_support/core_ext/string/access 663
	$B.45.4$ active_support/core_ext/string/acts_like $664$
	$B.45.5$ active_support/core_ext/string/conversions $664$
	$B.45.6$ active_support/core_ext/string/encoding $665$
	$B.45.7$ active_support/core_ext/string/exclude $665$
	B.45.8 active_support/core_ext/string/filters 665
	$B.45.9$ active_support/core_ext/string/inflections $666$
	$B.45.10$ active_support/core_ext/string/multibyte $669$
	$B.45.11 \ \texttt{active\_support/core\_ext/string/output\_safety}  670$
	B.45.12 active_support/core_ext/string/
	starts_ends_with 670
	B.45.13 active_support/core_ext/string/xchar 671
B.46	ActiveSupport::StringInquirer 671
B.47	Symbol 671
	B.47.1 active_support/json/encoding 671
B.48	ActiveSupport::Testing::Assertions 671
	B.48.1 active_support/testing/assertions 671
B.49	Time 673
	B.49.1 active_support/json/encoding 673
	B.49.2 active_support/core_ext/time/acts_like 673
	B.49.3 active_support/core_ext/time/calculations 673
	$\mathbf{D} \neq \mathbf{O} \neq $

B.49.4 active\_support/core\_ext/time/conversions 677

#### xxxii

#### Contents

	B.49.5	active_support/core_ext/time/marshal 679	
	B.49.6	active_support/core_ext/time/zones 679	
B.50	Active	eSupport::TimeWithZone 680	
B.51	ActiveSupport::TimeZone 681		
	B.51.1	active_support/values/time_zone 682	
B.52	Active	eSupport::TrueClass 684	
	B.52.1	active_support/core_ext/object/blank 684	
	B.52.2	active_support/json/encoding 684	
B.53	Active	eSupport::XmlMini 684	
	B.53.1	active_support/xml_mini 685	

#### Index

687

#### Method Index

697

### Foreword

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

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

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

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

> — David Heinemeier Hansson Creator of Ruby on Rails

This page intentionally left blank

### Foreword

From the beginning, the Rails framework turned web development on its head with the insight that the vast majority of time spent on projects amounted to meaningless sit-ups. Instead of having the time to think through your domain-specific code, you'd spend the first few weeks of a project deciding meaningless details. By making decisions for you, Rails frees you to kick off your project with a bang, getting a working prototype out the door quickly. This makes it possible to build an application with some meat on its bones in a few weekends, making Rails the web framework of choice for people with a great idea and a full-time job.

Rails makes some simple decisions for you, like what to name your controller actions and how to organize your directories. It also gets pretty aggressive, and sets developmentfriendly defaults for the database and caching layer you'll use, making it easy to change to more production-friendly options once you're ready to deploy.

By getting so aggressive, Rails makes it easy to put at least a few real users in front of your application within days, enabling you to start gathering the requirements from your users immediately, rather than spending months architecting a perfect solution, only to learn that your users use the application differently than you expected.

The Rails team built the Rails project itself according to very similar goals. Don't try to overthink the needs of your users. Get something out there that works, and improve it based on actual usage patterns. By all accounts, this strategy has been a smashing success, and with the blessing of the Rails core team, the Rails community leveraged the dynamism of Ruby to fill in the gaps in plugins. Without taking a close look at Rails, you might think that Rails' rapid prototyping powers are limited to the 15-minute blog demo, but that you'd fall off a cliff when writing a real app. This has never been true. In fact, in Rails 2.1, 2.2 and 2.3, the Rails team looked closely at common usage patterns

#### xxxvi

reflected in very popular plugins, adding features that would further reduce the number of sit-ups needed to start real-life applications.

By the release of Rails 2.3, the Rails ecosystem had thousands of plugins, and applications like Twitter started to push the boundaries of the Rails defaults. Increasingly, you might build your next Rails application using a non-relational database or deploy it inside a Java infrastructure using JRuby. It was time to take the tight integration of the Rails stack to the next level.

Over the course of 20 months, starting in January 2008, we looked at a wide range of plugins, spoke with the architects of some of the most popular Rails applications, and changed the way the Rails internals thought about its defaults.

Rather than start from scratch, trying to build a generic data layer for Rails, we took on the challenge of making it easy to give any ORM the same tight level of integration with the rest of the framework as Active Record. We accepted no compromises, taking the time to write the tight Active Record integration using the same APIs that we now expose for other ORMs. This covers the obvious, such as making it possible to generate a scaffold using DataMapper or Mongoid. It also covers the less obvious, such as giving alternative ORMs the same ability to include the amount of time spent in the model layer in the controller's log output.

We brought this philosophy to every area of Rails 3: flexibility without compromise. By looking at the ways that an estimated million developers use Rails, we could hone in on the needs of real developers and plugin authors, significantly improving the overall architecture of Rails based on real user feedback.

Because the Rails 3 internals are such a departure from what's come before, developers building long-lived applications and plugin developers need a resource that comprehensively covers the philosophy of the new version of the framework. *The Rails*<sup>TM</sup> 3 *Way* is a comprehensive resource that digs into the new features in Rails 3 and perhaps more importantly, the rationale behind them.

— Yehuda Katz Rails Core

## Introduction

As I write this new introduction in the spring of 2010, the official release of Rails 3.0 is looming, and what a big change it represents. The "Merb-ification" of Rails is almost complete! The new Rails is quite different from its predecessors in that its underlying architecture is more modular and elegant while increasing sheer performance significantly. The changes to Active Record are dramatic, with Arel's query method chaining replacing hashed find parameters that we were all used to.

There is a lot to love about Rails 3, and I do think that eventually most of the community will make the change. In most cases, I have not bothered to cover 2.x ways of doing things in Rails if they are significantly different from the Rails 3 way—hence the title change. I felt that naming the book "The Rails Way (Second Edition)" would be accurate, but possibly misleading. This new edition is a fully new book for a fully new framework. Practically every line of the book has been painstakingly revised and edited, with some fairly large chunks of the original book not making the new cut. It's taken well over a year, including six months of working every night to get this book done!

Even though Rails 3 is less opinionated than early versions, in that it allows for easy reconfiguration of Rails assumptions, this book is more opinionated than ever. The vast majority of Rails developers use RSpec, and I believe that is primarily because it is a superior choice to Test::Unit. Therefore, this book does not cover Test::Unit. I firmly believe that Haml is vastly, profoundly, better than ERb for view templating, so the book uses Haml exclusively.

#### 0.1 About This Book

This book is not a tutorial or basic introduction to Ruby or Rails. It is meant as a dayto-day reference for the full-time Rails developer. The more confident reader might be able to get started in Rails using just this book, extensive online resources, and his or her wits, but there are other publications that are more introductory in nature and might be a wee bit more appropriate for beginners.

Every contributor to this book works with Rails on a full-time basis. We do not spend our days writing books or training other people, although that is certainly something that we enjoy doing on the side.

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

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

#### 0.1.1 Book Structure

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

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

#### 0.1.2 Sample Code and Listings

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

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

Some of the source code for my examples can be found at http://github.com/ obie/tr3w\_time\_and\_expenses. Note that it is not a working nor complete application. It just made sense at times to keep the code in the context of an application and hopefully you might draw some inspiration from browsing it.

#### 0.1.3 Concerning Third-Party RubyGems and Plugins

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

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

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

Ruby on Rails is actually a vast ecosystem of core code, official plugins, and thirdparty plugins. That ecosystem has been exploding rapidly and provides all the raw technology you need to build even the most complicated enterprise-class web applications. My goal is to equip you with enough knowledge that you'll be able to avoid continuously reinventing the wheel.

#### 0.2 Recommended Reading and Resources

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

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

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

Ryan Bates does an excellent job explaining nuances of Rails development in his long-running series of free webcasts available at http://railscasts.com/

Last, but not least, this book's companion website at http://tr3w.com is the first place to look for reporting issues and finding additional resources, as they become available.

#### Regarding David Heinemeier Hansson, a.k.a. DHH

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

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

There are a number of notable people from the Rails world that are also referred to on a first-name basis in this book. Those include:

- Yehuda Katz
- Jamis Buck
- Xavier Noria

### 0.3 Goals

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

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

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

### 0.4 Prerequisites

The reader is assumed to have the following knowledge:

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

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

# CHAPTER 9 Advanced Active Record

Active Record is a simple object-relational mapping (ORM) framework compared to other popular ORM frameworks, such as Hibernate in the Java world. Don't let that fool you, though: Under its modest exterior, Active Record has some pretty advanced features. To really get the most effectiveness out of Rails development, you need to have more than a basic understanding of Active Record—things like knowing when to break out of the one-table/one-class pattern, or how to leverage Ruby modules to keep your code clean and free of duplication.

In this chapter, we wrap up this book's comprehensive coverage of Active Record by reviewing callbacks, observers, single-table inheritance (STI), and polymorphic models. We also review a little bit of information about metaprogramming and Ruby domain-specific languages (DSLs) as they relate to Active Record.

# 9.1 Scopes

Scopes (or "named scopes" if you're old school) allow you define and chain query criteria in a declarative and reusable manner.

```
class Timesheet < ActiveRecord::Base
  scope :submitted, where(:submitted => true)
  scope :underutilized, where('total_hours < 40')</pre>
```

To declare a scope, use the scope class method, passing it a name as a symbol and some sort of query definition. If your query is known at load time, you can simply use Arel criteria methods like where, order, and limit to construct the definition as shown in the example. On the other hand, if you won't have all the parameters for your query until runtime, use a lambda as the second parameter. It will get evaluated whenever the scope is invoked.

```
class User < ActiveRecord::Base
  scope :delinquent, lambda { where('timesheets_updated_at < ?',
1.week.ago)}</pre>
```

Invoke scopes as you would class methods.

```
>> User.delinquent
=> [#<User id: 2, timesheets_updated_at: "2010-01-07 01:56:29"...>]
```

#### 9.1.1 Scope Parameters

You can pass arguments to scope invocations by adding parameters to the lambda you use to define the scope query.

```
class BillableWeek < ActiveRecord::Base
  scope :newer_than, lambda { |date| where('start_date > ?', date) }
```

Then pass the argument to the scope as you would normally.

BillableWeek.newer\_than(Date.today)

#### 9.1.2 Chaining Scopes

One of the beauties of scopes is that you can chain them together to create complex queries from simple ones:

```
>> Timesheet.underutilized.submitted
=> [#<Timesheet id: 3, submitted: true, total_hours: 37 ...</pre>
```

Scopes can be chained together for reuse within scope definitions themselves. For instance, let's say that we always want to constrain the result set of underutilized to submitted timesheets:

```
class Timesheet < ActiveRecord::Base
scope :submitted, where(:submitted => true)
scope :underutilized, submitted.where('total_hours < 40')</pre>
```

#### 9.1.3 Scopes and has\_many

In addition to being available at the class context, scopes are available automatically on has\_many association attributes.

```
>> u = User.find 2
=> #<User id: 2, login: "obie"...>
>> u.timesheets.size
=> 3
>> u.timesheets.underutilized.size
=> 1
```

# 9.1.4 Scopes and Joins

You can use Arel's join method to create cross-model scopes. For instance, if we gave our recurring example Timesheet a submitted\_at date attribute instead of just a boolean, we could add a scope to User allowing us to see who is late on their timesheet submission.

```
scope :tardy, lambda {
   joins(:timesheets).
   where("timesheets.submitted_at <= ?", 7.days.ago).
   group("users.id")
}</pre>
```

Arel's to\_sql method is useful for debugging scope definitions and usage.

```
>> User.tardy.to_sql
=> "SELECT users.* FROM users
INNER JOIN timesheets ON timesheets.user_id = users.id
WHERE (timesheets.submitted_at <= '2010-07-06 15:27:05.117700')
GROUP BY users.id" # query formatted nicely for the book</pre>
```

Note that as demonstrated in the example, it's a good idea to use unambiguous column references (including table name) in cross-model scope definitions so that Arel doesn't get confused.

# 9.1.5 Scope Combinations

Our example of a cross-model scope violates good object-oriented design principles: it contains the logic for determining whether or not a Timesheet is submitted, which is code that properly belongs in the Timesheet class. Luckily we can use Arel's merge method (aliased as &) to fix it. First we put the late logic where it belongs, in Timesheet:

```
scope :late, lambda { where("timesheet.submitted_at <= ?", 7.days.ago) }</pre>
```

Then we use our new late scope in tardy:

```
scope :tardy, lambda {
   joins(:timesheets).group("users.id") & Timesheet.late
}
```

If you have trouble with this technique, make absolutely sure that your scopes' clauses refer to fully qualified column names. (In other words, don't forget to prefix column names with tables.) The console and to\_sql method is your friend for debugging.

#### 9.1.6 Default Scopes

There may arise use cases where you want certain conditions applied to the finders for your model. Consider our timesheet application has a default view of open timesheets—we can use a default scope to simplify our general queries.

```
class Timesheet < ActiveRecord::Base
  default_scope :where(:status => "open")
end
```

Now when we query for our Timesheets, by default the open condition will be applied:

```
>> Timesheet.all.map(&:status)
=> ["open", "open", "open"]
```

Default scopes also get applied to your models when building or creating them, which can be a great convenience or a nuisance if you are not careful. In our previous example, all new Timesheets will be created with a status of "open."

```
>> Timesheet.new
=> #<Timesheet id: nil, status: "open">
>> Timesheet.create
=> #<Timesheet id: 1, status: "open">
```

You can override this behavior by providing your own conditions or scope to override the default setting of the attributes.

```
>> Timesheet.where(:status => "new").new
=> #<Timesheet id: nil, status: "new">
>> Timesheet.where(:status => "new").create
=> #<Timesheet id: 1, status: "new">
```

There may be cases where at runtime you want to create a scope and pass it around as a first class object leveraging your default scope. In this case, Active Record provides the scoped method.

```
>> timesheets = Timesheet.scoped.order("submitted_at DESC")
=> [#<Timesheet id: 1, status: "open"]
>> timesheets.where(:name => "Durran Jordan")
=> []
```

There's another approach to scopes that provides a sleeker syntax, scoping, which allows the chaining of scopes via nesting within a block.

```
>> Timesheet.order("submitted_at DESC").scoping do
>> Timesheets.all
>> end
=> #<Timesheet id: 1, status: "open">
```

That's pretty nice, but what if we *don't* want our default scope to be included in our queries? In this case Active Record takes care of us through the unscoped method.

```
>> Timesheet.unscoped.order("submitted_at DESC")
=> [#<Timesheet id: 2, status: "submitted">]
```

Similarly to overriding our default scope with a relation when creating new objects, we can supply unscoped as well to remove the default attributes.

```
>> Timesheet.unscoped.new
=> #<Timesheet id: nil, status: nil>
```

## 9.1.7 Using Scopes for CRUD

You have a wide range of Active Record's CRUD methods available on scopes, which gives you some powerful abilities. For instance, let's give all our underutilized timesheets some extra hours.

```
>> u.timesheets.underutilized.collect(&:total_hours)
=> [37, 38]
>> u.timesheets.underutilized.update_all("total_hours = total_hours + 2")
=> 2
>> u.timesheets.underutilized.collect(&:total_hours)
=> [37, 38] # whoops, cached result
>> u.timesheets(true).underutilized.collect(&:total_hours)
=> [39] # results after telling association to reload
```

Scopes including a where clause using hashed conditions will populate attributes of objects built off of them with those attributes as default values. Admittedly it's a bit difficult to think of a plausible use case for this feature, but we'll show it in an example. First, we add the following scope to Timesheet:

```
scope :perfect, submitted.where(:total_hours => 40)
```

Now, building an object on the perfect scope should give us a submitted timesheet with 40 hours.

```
> Timesheet.perfect.build
=> #<Timesheet id: nil, submitted: true, user_id: nil, total_hours: 40
...>
```

As you've probably realized by now, the new Arel underpinnings of Active Record are tremendously powerful and truly elevate the Rails 3 platform.

# 9.2 Callbacks

This advanced feature of Active Record allows the savvy developer to attach behavior at a variety of different points along a model's life cycle, such as after initialization, before database records are inserted, updated or removed, and so on.

Callbacks can do a variety of tasks, ranging from simple things such as logging and massaging of attribute values prior to validation, to complex calculations. Callbacks can halt the execution of the life-cycle process taking place. Some callbacks can even modify the behavior of the model class on the fly. We'll cover all of those scenarios in this section, but first let's get a taste of what a callback looks like. Check out the following silly example:

```
class Beethoven < ActiveRecord::Base
before_destroy :last_words
protected
def last_words
   logger.info "Friends applaud, the comedy is over"
   end
end
```

So prior to dying (ehrm, being destroy'd), the last words of the Beethoven class will always be logged for posterity. As we'll see soon, there are 14 different opportunities to add behavior to your model in this fashion. Before we get to that list, let's cover the mechanics of registering a callback.

#### 9.2.1 Callback Registration

Overall, the most common way to register a callback method is to declare it at the top of the class using a typical Rails macro-style class method. However, there's a less verbose way to do it also. Simply implement the callback as a method in your class. In other words, I could have coded the prior example as follows:

```
class Beethoven < ActiveRecord::Base
  protected
  def before_destroy
    logger.info "Friends applaud, the comedy is over"
  end
end</pre>
```

This is a rare case of the less-verbose solution being bad. In fact, it is almost always preferable, dare I say it is the Rails way, to use the callback macros over implementing

callback methods, for the following reasons:

- Macro-style callback declarations are added near the top of the class definition, making the existence of that callback more evident versus a method body potentially buried later in the file.
- Macro-style callbacks add callback methods to a queue. That means that more than one method can be hooked into the same slot in the life cycle. Callbacks will be invoked in the order in which they were added to the queue.
- Callback methods for the same hook can be added to their queue at different levels of an inheritance hierarchy and still work—they won't override each other the way that methods would.
- Callbacks defined as methods on the model are always called last.

## 9.2.2 One-Liners

Now, if (and only if) your callback routine is really short,<sup>1</sup> you can add it by passing a block to the callback macro. We're talking one-liners!

```
class Napoleon < ActiveRecord::Base
  before_destroy { logger.info "Josephine..." }
   ...
end</pre>
```

As of Rails 3, the block passed to a callback is executed via instance\_eval so that its scope is the record itself (versus needing to act on a passed in record variable). The following example implements "paranoid" model behavior, covered later in the chapter.

```
class Account < ActiveRecord::Base
before_destroy { update_attribute(:deleted_at, Time.now); false }
...
```

#### 9.2.3 Protected or Private

Except when you're using a block, the access level for callback methods should always be protected or private. It should never be public, since callbacks should never be called from code outside the model.

<sup>1.</sup> If you are browsing old Rails source code, you might come across callback macros receiving a short string of Ruby code to be evaluated in the binding of the model object. That way of adding callbacks was deprecated in Rails 1.2, because you're always better off using a block in those situations.

Believe it or not, there are even more ways to implement callbacks, but we'll cover those techniques further along in the chapter. For now, let's look at the lists of callback hooks available.

# 9.2.4 Matched before/after Callbacks

In total, there are 14 types of callbacks you can register on your models! Twelve of them are matching before/after callback pairs, such as before\_validation and after\_validation. (The other two, after\_initialize and after\_find, are special, and we'll discuss them later in this section.)

#### List of Callbacks

This is the list of callback hooks available during a save operation. (The list varies slightly depending on whether you're saving a new or existing record.)

- before\_validation
- before\_validation\_on\_create
- after\_validation
- after\_validation\_on\_create
- before\_save
- before\_create (for new records) and before\_update (for existing records)
- (Database actually gets an INSERT or UPDATE statement here)
- after\_create (for new records) and after\_update (for existing records)
- after\_save

Delete operations have their own two callbacks:

- before\_destroy
- (Database actually gets a DELETE statement here)
- after\_destroy is called after all attributes have been frozen (read-only)

Additionally transactions have callbacks as well, for when you want actions to occur after the database is guaranteed to be in a permanent state. Note that only "after" callbacks exist here because of the nature of transactions—it's a bad idea to be able to interfere with the actual operation itself.

- after\_commit
- after\_commit\_on\_create
- after\_commit\_on\_update
- after\_commit\_on\_destroy
- after\_rollback
- after\_rollback\_on\_create
- after\_rollback\_on\_update
- after\_rollback\_on\_destroy

# 9.2.5 Halting Execution

If you return a boolean false (not nil) from a callback method, Active Record halts the execution chain. No further callbacks are executed. The save method will return false, and save! will raise a RecordNotSaved error.

Keep in mind that because the last expression of a Ruby method is returned implicitly, it is a pretty common bug to write a callback that halts execution unintentionally. If you have an object with callbacks that mysteriously fails to save, make sure you aren't returning false by mistake.

## 9.2.6 Callback Usages

Of course, the callback you should use for a given situation depends on what you're trying to accomplish. The best I can do is to serve up some examples to inspire you with your own code.

Cleaning Up Attribute Formatting with **before\_validate\_on\_create** 

The most common examples of using before\_validation callbacks have to do with cleaning up user-entered attributes. For example, the following CreditCard class cleans up its number attribute so that false negatives don't occur on validation:

```
class CreditCard < ActiveRecord::Base
...
def before_validation_on_create
   # Strip everything in the number except digits
   self.number = number.gsub(/[^0-9]/, "")
end
end</pre>
```

#### Geocoding with **before\_save**

Assume that you have an application that tracks addresses and has mapping features. Addresses should always be geocoded before saving, so that they can be displayed rapidly on a map later.<sup>2</sup>

As is often the case, the wording of the requirement itself points you in the direction of the before\_save callback:

```
class Address < ActiveRecord::Base
include GeoKit::Geocoders
before_save :geolocate
validates_presence_of :street, :city, :state, :zip
...
def to_s
    "#{street} #{city}, #{state} #{zip}"
end
protected
def geolocate
    res = GoogleGeocoder.geocode(to_s)
    self.latitude = res.lat
    self.longitude = res.lng
end
end
```

Before we move on, there are a couple of additional considerations. The preceding code works great if the geocoding succeeds, but what if it doesn't? Do we still want to allow the record to be saved? If not, we should halt the execution chain:

```
def geolocate
  res = GoogleGeocoder.geocode(to_s)
  return false if not res.success  # halt execution
  self.latitude = res.lat
  self.longitude = res.lng
end
```

The only problem remaining is that we give the rest of our code (and by extension, the end user) no indication of why the chain was halted. Even though we're not in a validation routine, I think we can put the errors collection to good use here:

```
def geolocate
  res = GoogleGeocoder.geocode(to_s)
```

<sup>2.</sup> I recommend the excellent GeoKit for Rails plugin available at http://geokit.rubyforge.org/.

```
if res.success
   self.latitude = res.lat
   self.longitude = res.lng
   else
    errors[:base] << "Geocoding failed. Please check address."
   return false
   end
end</pre>
```

If the geocoding fails, we add a base error message (for the whole object) and halt execution, so that the record is not saved.

#### Exercise Your Paranoia with **before\_destroy**

What if your application has to handle important kinds of data that, once entered, should never be deleted? Perhaps it would make sense to hook into Active Record's destroy mechanism and somehow mark the record as deleted instead?

The following example depends on the accounts table having a deleted\_at date-time column.

```
class Account < ActiveRecord::Base
...
def before_destroy
   update_attribute(:deleted_at, Time.now)
   false
end</pre>
```

end

I chose to implement it as a callback method so that I am guaranteed it will execute last in the before\_destroy queue. It returns false so that execution is halted and the underlying record is not actually deleted from the database.<sup>3</sup>

It's probably worth mentioning that there are ways that Rails allows you to unintentionally circumvent before\_destroy callbacks:

• The delete and delete\_all class methods of ActiveRecord::Base are almost identical. They remove rows directly from the database without instantiating the corresponding model instances, which means no callbacks will occur.

<sup>3.</sup> Real-life implementation of the example would also need to modify all finders to include deleted\_at is NULL conditions; otherwise, the records marked deleted would continue to show up in the application. That's not a trivial undertaking, and luckily you don't need to do it yourself. There's a Rails plugin named ActsAsParanoid by Rick Olson that does exactly that, and you can find it at http://svn.techno-weenie.net/projects/plugins/acts\_as\_paranoid.

• Model objects in associations defined with the option :dependent => :delete\_ all will be deleted directly from the database when removed from the collection using the association's clear or delete methods.

#### Cleaning Up Associated Files with after\_destroy

Model objects that have files associated with them, such as attachment records and uploaded images, can clean up after themselves when deleted using the after\_destroy callback. The following method from Rick Olson's old AttachmentFu<sup>4</sup> plugin is a good example:

```
# Destroys the file. Called in the after_destroy callback
def destroy_file
FileUtils.rm(full_filename)
...
rescue
logger.info "Exception destroying #{full_filename ... }"
logger.warn $!.backtrace.collect { |b| " > #{b}" }.join("\n")
end
```

# 9.2.7 Special Callbacks: **after\_initialize** and **after\_find**

The after\_initialize callback is invoked whenever a new Active Record model is instantiated (either from scratch or from the database). Having it available prevents you from having to muck around with overriding the actual initialize method.

The after\_find callback is invoked whenever Active Record loads a model object from the database, and is actually called before after\_initialize, if both are implemented. Because after\_find and after\_initialize are called for each object found and instantiated by finders, performance constraints dictate that they can only be added as methods, and not via the callback macros.

What if you want to run some code only the first time that a model is ever instantiated, and not after each database load? There is no native callback for that scenario, but you can do it using the after\_initialize callback. Just add a condition that checks to see if it is a new record:

```
def after_initialize
    if new?
        ...
    end
end
```

<sup>4.</sup> Get AttachmentFu at http://svn.techno-weenie.net/projects/plugins/attachment\_fu.

In a number of Rails apps that I've written, I've found it useful to capture user preferences in a serialized hash associated with the User object. The serialize feature of Active Record models makes this possible, since it transparently persists Ruby object graphs to a text column in the database. Unfortunately, you can't pass it a default value, so I have to set one myself:

```
class User < ActiveRecord::Base
  serialize :preferences # defaults to nil
  ...
  protected
    def after_initialize
       self.preferences ||= Hash.new
    end
end</pre>
```

Using the after\_initialize callback, I can automatically populate the preferences attribute of my user model with an empty hash, so that I never have to worry about it being nil when I access it with code such as user.preferences[:show\_help\_text] = false.

Ruby's metaprogramming capabilities combined with the ability to run code whenever a model is loaded using the after\_find callback are a powerful mix. Since we're not done learning about callbacks yet, we'll come back to uses of after\_find later on in the chapter, in the section "Modifying Active Record Classes at Runtime."

## 9.2.8 Callback Classes

It is common enough to want to reuse callback code for more than one object that Rails gives you a way to write callback classes. All you have to do is pass a given callback queue an object that responds to the name of the callback and takes the model object as a parameter.

Here's our paranoid example from the previous section as a callback class:

```
class MarkDeleted
  def self.before_destroy(model)
    model.update_attribute(:deleted_at, Time.now)
    return false
    end
end
```

The behavior of MarkDeleted is stateless, so I added the callback as a class method. Now you don't have to instantiate MarkDeleted objects for no good reason. All you

do is pass the class to the callback queue for whichever models you want to have the mark-deleted behavior:

```
class Account < ActiveRecord::Base
  before_destroy MarkDeleted
  ...
end
class Invoice < ActiveRecord::Base
  before_destroy MarkDeleted
  ...
end
```

#### Multiple Callback Methods in One Class

There's no rule that says you can't have more than one callback method in a callback class. For example, you might have special audit log requirements to implement:

```
class Auditor
  def initialize(audit_log)
    @audit_log = audit_log
  end
  def after_create(model)
    @audit_log.created(model.inspect)
  end
  def after_update(model)
    @audit_log.updated(model.inspect)
  end
  def after_destroy(model)
    @audit_log.destroyed(model.inspect)
  end
end
```

To add audit logging to an Active Record class, you would do the following:

```
class Account < ActiveRecord::Base
  after_create Auditor.new(DEFAULT_AUDIT_LOG)
  after_update Auditor.new(DEFAULT_AUDIT_LOG)
  after_destroy Auditor.new(DEFAULT_AUDIT_LOG)
  ...
end
```

Wow, that's ugly, having to add three Auditors on three lines. We could extract a local variable called auditor, but it would still be repetitive. This might be an opportunity to take advantage of Ruby's open classes, the fact that you can modify classes that aren't part of your application.

Wouldn't it be better to simply say acts\_as\_audited at the top of the model that needs auditing? We can quickly add it to the ActiveRecord::Base class, so that it's available for all our models.

On my projects, the file where "quick and dirty" code like the method in Listing 9.1 would reside is lib/core\_ext/active\_record\_base.rb, but you can put it anywhere you want. You could even make it a plugin (as detailed in Chapter 19, "Extending Rails with Plugins").

#### Listing 9.1 A quick-and-dirty "acts as audited" method

```
class ActiveRecord::Base
  def self.acts_as_audited(audit_log=DEFAULT_AUDIT_LOG)
    auditor = Auditor.new(audit_log)
    after_create auditor
    after_update auditor
    after_destroy auditor
    end
end
```

Now, the top of Account is a lot less cluttered:

```
class Account < ActiveRecord::Base
  acts_as_audited
```

#### Testability

When you add callback methods to a model class, you pretty much have to test that they're functioning correctly in conjunction with the model to which they are added. That may or may not be a problem. In contrast, callback classes are super-easy to test in isolation.

```
def test_auditor_logs_created
  (model = mock).expects(:inspect).returns('foo')
  (log = mock).expects(:created).with('foo')
  Auditor.new(log).after_create(model)
end
```

# 9.3 Calculation Methods

All Active Record classes have a calculate method that provides easy access to aggregate function queries in the database. Methods for count, sum, average, minimum, and maximum have been added as convenient shortcuts.

Options such as conditions, :order, :group, :having, and :joins can be passed to customize the query.

There are two basic forms of output:

- **Single aggregate value** The single value is type cast to Fixnum for COUNT, Float for AVG, and the given column's type for everything else.
- Grouped values This returns an ordered hash of the values and groups them by the :group option. It takes either a column name, or the name of a belongs\_to association.

The following options are available to all calculation methods:

- :conditions An SQL fragment like "administrator = 1" or [ "user\_name =
   ?", username ]. See conditions in the intro to ActiveRecord::Base.
- **:include** Eager loading, see Associations for details. Since calculations don't load anything, the purpose of this is to access fields on joined tables in your conditions, order, or group clauses.
- :joins An SQL fragment for additional joins like "LEFT JOIN comments ON comments.post\_id = id". (Rarely needed). The records will be returned read-only since they will have attributes that do not correspond to the table's columns.
- **:order** An SQL fragment like "created\_at DESC, name" (really only used with GROUP BY calculations).
- **: group** An attribute name by which the result should be grouped. Uses the GROUP BY SQL-clause.
- **:select** By default, this is \* as in SELECT \* FROM, but can be changed if you, for example, want to do a join, but not include the joined columns.
- :distinct Set this to true to make this a distinct calculation, such as SELECT COUNT(DISTINCT posts.id) ...

The following examples illustrate the usage of various calculation methods.

Person.calculate(:count, :all) # The same as Person.count # SELECT AVG(age) FROM people Person.average(:age) # Selects the minimum age for everyone with a last name other than 'Drake' Person.minimum(:age).where('last\_name <> ?', 'Drake')

```
# Selects the minimum age for any family without any minors
Person.minimum(:age).having('min(age) > 17').group(:last_name)
```

## 9.3.1 average(column\_name, \*options)

Calculates the average value on a given column. The first parameter should be a symbol identifying the column to be averaged.

#### 9.3.2 count(column\_name, \*options)

Count operates using three different approaches. Count without parameters will return a count of all the rows for the model. Count with a column\_name will return a count of all the rows for the model with the supplied colum present. Lastly, count using :options will find the row count matched by the options used. In the last case you would send an options hash as the only parameter. 213

```
total_contacts = person.contacts.count(:from => "contact_cards")
```

Options are the same as with all other calculations methods with the additional option of :from which is by default the name of the table name of the class, however it can be changed to a different table name or even that of a database view. Remember that Person.count(:all) will not work because :all will be treated as a condition, you should use Person.count instead.

## 9.3.3 maximum(column\_name, \*options)

Calculates the maximum value on a given column. The first parameter should be a symbol identifying the column to be calculated.

# 9.3.4 minimum(column\_name, \*options)

Calculates the minimum value on a given column. The first parameter should be a symbol identifying the column to be calculated.

## 9.3.5 sum(column\_name, \*options)

Calculates a summed value in the database using SQL. The first parameter should be a symbol identifying the column to be summed.

## 9.4 Observers

The single responsibility principle is a very important tenet of object-oriented programming. It compels us to keep a class focused on a single concern. As you've learned in the previous section, callbacks are a useful feature of Active Record models that allow us to hook in behavior at various points of a model object's life cycle. Even if we pull that extra behavior out into callback classes, the hook still requires code changes in the model class definition itself. On the other hand, Active Record gives us a way to hook in to models that is completely transparent: Observers.

Here is the functionality of our old Auditor callback class as an observer of Account objects:

```
class AccountObserver < ActiveRecord::Observer
  def after_create(model)
        DEFAULT_AUDIT_LOG.created(model.inspect)
  end
  def after_update(model)
        DEFAULT_AUDIT_LOG.updated(model.inspect)
  end
  def after_destroy(model)
        DEFAULT_AUDIT_LOG.destroyed(model.inspect)
  end
end
```

## 9.4.1 Naming Conventions

When ActiveRecord::Observer is subclassed, it breaks down the name of the subclass by stripping off the "Observer" part. In the case of our AccountObserver in the preceding example, it would know that you want to observe the Account class. However, that's not always desirable behavior. In fact, with general-purpose code such as our Auditor, it's positively a step backward, so it is possible to overrule the naming convention with the use of the observe macro-style method. We still extend ActiveRecord::Observer, but we can call the subclass whatever we want and tell it explicitly what to observe using the observe method, which accepts one or more arguments.

```
class Auditor < ActiveRecord::Observer
observe Account, Invoice, Payment
def after_create(model)
    DEFAULT_AUDIT_LOG.created(model.inspect)
end</pre>
```

```
def after_update(model)
    DEFAULT_AUDIT_LOG.updated(model.inspect)
    end
    def after_destroy(model)
    DEFAULT_AUDIT_LOG.destroyed(model.inspect)
    end
end
```

#### 9.4.2 Registration of Observers

If there weren't a place for you to tell Rails which observers to load, they would never get loaded at all, since they're not referenced from any other code in your application. Register observers with the following kind of code in an initializer:

```
# Activate observers that should always be running
ActiveRecord::Base.observers = Auditor
```

#### 9.4.3 Timing

Observers are notified after the in-object callbacks are triggered.<sup>5</sup> It's not possible to act on the whole object from an observer without having the object's own callbacks executed first.

```
Durran says . . .
```

For those of us who love to be organized, you can now put your observers in a separate directory under app if your heart desires. You won't need to perform custom loading anymore since Rails now loads all files under the app directory automatically.

# 9.5 Single-Table Inheritance (STI)

A lot of applications start out with a User model of some sort. Over time, as different kinds of users emerge, it might make sense to make a greater distinction between them. Admin and Guest classes are introduced, as subclasses of User. Now, the shared behavior can reside in User, and subtype behavior can be pushed down to subclasses. However, all user data can still reside in the users table—all you need to do is introduce a type column that will hold the name of the class to be instantiated for a given row.

<sup>5.</sup> https://rails.lighthouseapp.com/projects/8994/tickets/230 contains an interesting discussion about callback execution order.

To continue explaining single-table inheritance, let's turn back to our example of a recurring Timesheet class. We need to know how many billable\_hours are outstanding for a given user. The calculation can be implemented in various ways, but in this case we've chosen to write a pair of class and instance methods on the Timesheet class:

```
class Timesheet < ActiveRecord::Base
...
def billable_hours_outstanding
  if submitted?
    billable_weeks.map(&:total_hours).sum
  else
    0
  end
end
def self.billable_hours_outstanding_for(user)
    user.timesheets.map(&:billable_hours_outstanding).sum
end
```

end

I'm not suggesting that this is good code. It works, but it's inefficient and that if/else condition is a little fishy. Its shortcomings become apparent once requirements emerge about marking a Timesheet as paid. It forces us to modify Timesheet's billable\_hours\_outstanding method again:

```
def billable_hours_outstanding
  if submitted? && not paid?
    billable_weeks.map(&:total_hours).sum
  else
    0
  end
end
```

That latest change is a clear violation of the open-closed principle,<sup>6</sup> which urges you to write code that is open for extension, but closed for modification. We know that we violated the principle, because we were forced to change the billable\_hours\_outstanding method to accommodate the new Timesheet status. Though it may not seem like a large problem in our simple example, consider the amount of conditional code that will end up in the Timesheet class once we start having to implement functionality such as paid\_hours and unsubmitted\_hours.

<sup>6.</sup> http://en.wikipedia.org/wiki/Open/closed\_principle has a good summary.

So what's the answer to this messy question of the constantly changing conditional? Given that you're reading the section of the book about single-table inheritance, it's probably no big surprise that we think one good answer is to use object-oriented inheritance. To do so, let's break our original Timesheet class into four classes.

```
class Timesheet < ActiveRecord::Base
  # non-relevant code ommitted
  def self.billable hours outstanding for(user)
    user.timesheets.map(&:billable_hours_outstanding).sum
  end
end
class DraftTimesheet < Timesheet
  def billable_hours_outstanding
    0
  end
end
class SubmittedTimesheet < Timesheet
  def billable hours outstanding
    billable_weeks.map(&:total_hours).sum
  end
end
```

Now when the requirements demand the ability to calculate partially paid timesheets, we need only add some behavior to a PaidTimesheet class. No messy conditional statements in sight!

```
class PaidTimesheet < Timesheet
  def billable_hours_outstanding
    billable_weeks.map(&:total_hours).sum - paid_hours
end
end</pre>
```

#### 9.5.1 Mapping Inheritance to the Database

Mapping object inheritance effectively to a relational database is not one of those problems with a definitive solution. We're only going to talk about the one mapping strategy that Rails supports natively, which is single-table inheritance, called STI for short.

In STI, you establish one table in the database to holds all of the records for any object in a given inheritance hierarchy. In Active Record STI, that one table is named after the top parent class of the hierarchy. In the example we've been considering, that table would be named timesheets.

Hey, that's what it was called before, right? Yes, but to enable STI we have to add a type column to contain a string representing the type of the stored object. The following migration would properly set up the database for our example:

```
class AddTypeToTimesheet < ActiveRecord::Migration
  def self.up
    add_column :timesheets, :type, :string
  end
  def self.down
    remove_column :timesheets, :type
  end
end</pre>
```

No default value is needed. Once the type column is added to an Active Record model, Rails will automatically take care of keeping it populated with the right value. Using the console, we can see this behavior in action:

```
>> d = DraftTimesheet.create
>> d.type
=> 'DraftTimesheet'
```

When you try to find an object using the find methods of a base STI class, Rails will automatically instantiate objects using the appropriate subclass. This is especially useful in polymorphic situations, such as the timesheet example we've been describing, where we retrieve all the records for a particular user and then call methods that behave differently depending on the object's class.

```
>> Timesheet.find(:first)
=> #<DraftTimesheet:0x2212354...>
```

Sebastian says . . .

The word "type" is a very common column name and you might have plenty of uses for it not related to STI—which is why it's very likely you've experienced an ActiveRecord::SubclassNotFound error. Rails will read the "type" column of your Car class and try to find an "SUV" class that doesn't exist. The solution is simple: Tell Rails to use another column for STI with the following code:

#### Note

Rails won't complain about the missing column; it will simply ignore it. Recently, the error message was reworded with a better explanation, but too many developers skim error messages and then spend an hour trying to figure out what's wrong with their models. (A lot of people skim sidebar columns too when reading books, but hey, at least I am doubling their chances of learning about this problem.)

#### 9.5.2 STI Considerations

Although Rails makes it extremely simple to use single-table inheritance, there are a few caveats that you should keep in mind.

To begin with, you cannot have an attribute on two different subclasses with the same name but a different type. Since Rails uses one table to store all subclasses, these attributes with the same name occupy the same column in the table. Frankly, there's not much of a reason why that should be a problem unless you've made some pretty bad data-modeling decisions.

More importantly, you need to have one column per attribute on any subclass and any attribute that is not shared by all the subclasses must accept nil values. In the recurring example, PaidTimesheet has a paid\_hours column that is not used by any of the other subclasses. DraftTimesheet and SubmittedTimesheet will not use the paid\_hours column and leave it as null in the database. In order to validate data for columns not shared by all subclasses, you must use Active Record validations and not the database.

Third, it is not a good idea to have subclasses with too many unique attributes. If you do, you will have one database table with many null values in it. Normally, a tree of subclasses with a large number of unique attributes suggests that something is wrong with your application design and that you should refactor. If you have an STI table that is getting out of hand, it is time to reconsider your decision to use inheritance to solve your particular problem. Perhaps your base class is too abstract?

Finally, legacy database constraints may require a different name in the database for the type column. In this case, you can set the new column name using the class method set\_inheritance\_column in the base class. For the Timesheet example, we could do the following:

```
class Timesheet < ActiveRecord::Base
   set_inheritance_column 'object_type'
end</pre>
```

Now Rails will automatically populate the object\_type column with the object's type.

#### 9.5.3 STI and Associations

It seems pretty common for applications, particularly data-management ones, to have models that are very similar in terms of their data payload, mostly varying in their behavior and associations to each other. If you used object-oriented languages prior to Rails, you're probably already accustomed to breaking down problem domains into hierarchical structures.

Take for instance, a Rails application that deals with the population of states, counties, cities, and neighborhoods. All of these are places, which might lead you to define an STI class named Place as shown in Listing 9.2. I've also included the database schema for clarity:<sup>7</sup>

Listing 9.2 The places database schema and the place class

```
== Schema Information
#
 Table name: places
#
  id :integer(11) not null, primary key
#
  region id :integer(11)
#
  type :string(255)
#
#
  name :string(255)
#
  description :string(255)
  latitude :decimal(20, 1)
#
# longitude :decimal(20, 1)
# population :integer(11)
# created_at :datetime
# updated_at :datetime
class Place < ActiveRecord::Base
end
```

Place is in essence an abstract class. It should not be instantiated, but there is no foolproof way to enforce that in Ruby. (No big deal, this isn't Java!) Now let's go ahead

<sup>7.</sup> For autogenerated schema information added to the top of your model classes, try Dave Thomas's annotate\_models plugin at http://svn.pragprog.com/Public/plugins/

and define concrete subclasses of Place:

```
class State < Place
has_many :counties, :foreign_key => 'region_id'
end
class County < Place
  belongs_to :state, :foreign_key => 'region _id'
  has_many :cities, :foreign_key => 'region _id'
end
class City < Place
  belongs_to :county, :foreign_key => 'region _id'
end
```

You might be tempted to try adding a cities association to State, knowing that has\_many :through works with both belongs\_to and has\_many target associations. It would make the State class look something like this:

```
class State < Place
has_many :counties, :foreign_key => 'region_id'
has_many :cities, :through => :counties
end
```

That would certainly be cool, if it worked. Unfortunately, in this particular case, since there's only one underlying table that we're querying, there simply isn't a way to distinguish among the different kinds of objects in the query:

```
Mysql::Error: Not unique table/alias: 'places': SELECT places.* FROM
places INNER JOIN places ON places.region_id = places.id WHERE
((places.region_id = 187912) AND ((places.type = 'County'))) AND
((places.`type` = 'City' ))
```

What would we have to do to make it work? Well, the most realistic would be to use specific foreign keys, instead of trying to overload the meaning of region\_id for all the subclasses. For starters, the places table would look like the example in Listing 9.3.

```
Listing 9.3 The places database schema revised
```

```
# == Schema Information
#
# Table name: places
#
# id :integer(11) not null, primary key
# state_id :integer(11)
# county_id :integer(11)
# type :string(255)
# name :string(255)
# description :string(255)
```

```
# latitude :decimal(20, 1)
# longitude :decimal(20, 1)
# population :integer(11)
# created_at :datetime
# updated_at :datetime
```

The subclasses would be simpler without the :foreign\_key options on the associations. Plus you could use a regular has\_many relationship from State to City, instead of the more complicated has\_many :through.

```
class State < Place
has_many :counties
has_many :cities
end
class County < Place
belongs_to :state
has_many :cities
end
class City < Place
belongs_to :county
end
```

Of course, all those null columns in the places table won't win you any friends with relational database purists. That's nothing, though. Just a little bit later in this chapter we'll take a second, more in-depth look at polymorphic has\_many relationships, which will make the purists positively hate you.

# 9.6 Abstract Base Model Classes

In contrast to single-table inheritance, it is possible for Active Record models to share common code via inheritance and still be persisted to different database tables. In fact, every Rails developer uses an abstract model in their code whether they realize it or not: ActiveRecord::Base.<sup>8</sup>

The technique involves creating an abstract base model class that persistent subclasses will extend. It's actually one of the simpler techniques that we broach in this chapter. Let's take the Place class from the previous section (refer to Listing 9.3) and revise it to

<sup>8.</sup> http://m.onkey.org/2007/12/9/namespaced-models

be an abstract base class in Listing 9.4. It's simple really—we just have to add one line of code:

Listing 9.4 The abstract place class

```
class Place < ActiveRecord::Base
  self.abstract_class = true
end</pre>
```

Marking an Active Record model abstract is essentially the opposite of making it an STI class with a type column. You're telling Rails: "Hey, I don't want you to assume that there is a table named places."

In our running example, it means we would have to establish tables for states, counties, and cities, which might be exactly what we want. Remember though, that we would no longer be able to query across subtypes with code like Place.all.

Abstract classes is an area of Rails where there aren't too many hard-and-fast rules to guide you—experience and gut feeling will help you out.

In case you haven't noticed yet, both class and instance methods are shared down the inheritance hierarchy of Active Record models. So are constants and other class members brought in through module inclusion. That means we can put all sorts of code inside Place that will be useful to its subclasses.

## 9.7 Polymorphic has\_many Relationships

Rails gives you the ability to make one class belong\_to more than one type of another class, as eloquently stated by blogger Mike Bayer:

The "polymorphic association," on the other hand, while it bears some resemblance to the regular polymorphic union of a class hierarchy, is not really the same since you're only dealing with a particular association to a single target class from any number of source classes, source classes which don't have anything else to do with each other; i.e., they aren't in any particular inheritance relationship and probably are all persisted in completely different tables. In this way, the polymorphic association has a lot less to do with object inheritance and a lot more to do with aspect-oriented programming (AOP); a particular concept needs to be applied to a divergent set of entities which otherwise are not directly related. Such a concept is referred to as a cross-cutting concern, such as, all the entities in your domain need to support a history log of all changes to a common logging table. In the AR example, an Order and a User object are illustrated to both require links to an Address object.<sup>9</sup>

<sup>9.</sup> http://techspot.zzzeek.org/?p=13

In other words, this is not polymorphism in the typical object-oriented sense of the word; rather, it is something unique to Rails.

#### 9.7.1 In the Case of Models with Comments

In our recurring Time and Expenses example, let's assume that we want both BillableWeek and Timesheet to have many comments (a shared Comment class). A naive way to solve this problem might be to have the Comment class belong to both the BillableWeek and Timesheet classes and have billable\_week\_id and timesheet\_id as columns in its database table.

```
class Comment < ActiveRecord::Base
  belongs_to :timesheet
  belongs_to :expense_report
end
```

I call that approach is naive because it would be difficult to work with and hard to extend. Among other things, you would need to add code to the application to ensure that a Comment never belonged to both a BillableWeek and a Timesheet at the same time. The code to figure out what a given comment is attached to would be cumbersome to write. Even worse, every time you want to be able to add comments to another type of class, you'd have to add another nullable foreign key column to the comments table.

Rails solves this problem in an elegant fashion, by allowing us to define what it terms polymorphic associations, which we covered when we described the :polymorphic => true option of the belongs\_to association in Chapter 7, Active Record Associations.

#### The Interface

Using a polymorphic association, we need define only a single belongs\_to and add a pair of related columns to the underlying database table. From that moment on, any class in our system can have comments attached to it (which would make it commentable), without needing to alter the database schema or the Comment model itself.

```
class Comment < ActiveRecord::Base
    belongs_to :commentable, :polymorphic => true
end
```

There isn't a Commentable class (or module) in our application. We named the association :commentable because it accurately describes the interface of objects that will be associated in this way. The name : commentable will turn up again on the other side of the association:

```
class Timesheet < ActiveRecord::Base
has_many :comments, :as => :commentable
end
class BillableWeek < ActiveRecord::Base
has_many :comments, :as => :commentable
end
```

Here we have the friendly has\_many association using the :as option. The :as marks this association as polymorphic, and specifies which interface we are using on the other side of the association. While we're on the subject, the other end of a polymorphic belongs\_to can be either a has\_many or a has\_one and work identically.

#### The Database Columns

Here's a migration that will create the comments table:

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
    t.text :body
    t.integer :commentable
    t.string :commentable_type
    end
    end
end</pre>
```

As you can see, there is a column called commentable\_type, which stores the class name of associated object. The Migrations API actually gives you a one-line shortcut with the references method, which takes a polymorphic option:

```
create_table :comments do |t|
   t.text :body
   t.references :commentable, :polymorphic => true
end
```

We can see how it comes together using the Rails console (some lines ommitted for brevity):

```
>> c = Comment.create(:text => "I could be commenting anything.")
>> t = TimeSheet.create
>> b = BillableWeek.create
>> c.update_attribute(:commentable, t)
=> true
>> "#{c.commentable_type}: #{c.commentable_id}"
=> "Timesheet: 1"
```

```
>> c.update_attribute(:commentable, b)
=> true
>> "#{c.commentable_type}: #{c.commentable_id}"
=> "BillableWeek: 1"
```

As you can tell, both the Timesheet and the BillableWeek that we played with in the console had the same id (1). Thanks to the commentable\_type attribute, stored as a string, Rails can figure out which is the correct related object.

#### has\_many :through and Polymorphics

There are some logical limitations that come into play with polymorphic associations. For instance, since it is impossible for Rails to know the tables necessary to join through a polymorphic association, the following hypothetical code, which tries to find everything that the user has commented on, will not work.

```
class Comment < ActiveRecord::Base
  belongs_to :user # author of the comment
  belongs_to :commentable, :polymorphic => true
end
class User < ActiveRecord::Base
  has_many :comments
  has_many :commentables, :through => :comments
end
>> User.first.comments
ActiveRecord::HasManyThroughAssociationPolymorphicError: Cannot have
a has_many :through association 'User#commentables' on the polymorphic
object 'Comment#commentable'.
```

If you really need it, has\_many :through is possible with polymorphic associations, but only by specifying exactly what type of polymorphic associations you want. To do so, you must use the :source\_type option. In most cases, you will also need to use the :source option, since the association name will not match the interface name used for the polymorphic association:

```
class User < ActiveRecord::Base
has_many :comments
has_many :commented_timesheets, :through => :comments,
                :source => :commentable, :source_type => 'Timesheet'
has_many :commented_billable_weeks, :through => :comments,
                :source => :commentable, :source_type => 'BillableWeek'
end
```

It's verbose, and the whole scheme loses its elegance if you go this route, but it works:

```
>> User.first.commented_timesheets
=> [#<Timesheet ...>]
```

# 9.8 Foreign-key Constraints

As we work toward the end of this book's coverage of Active Record, you might have noticed that we haven't really touched on a subject of particular importance to many programmers: foreign-key constraints in the database. That's mainly because use of foreign-key constraints simply isn't the Rails way to tackle the problem of relational integrity. To put it mildly, that opinion is controversial and some developers have written off Rails (and its authors) for expressing it.

There really isn't anything stopping you from adding foreign-key constraints to your database tables, although you'd do well to wait until after the bulk of development is done. The exception, of course, is those polymorphic associations, which are probably the most extreme manifestation of the Rails opinion against foreign-key constraints. Unless you're armed for battle, you might not want to broach that particular subject with your DBA.

# 9.9 Using Value Objects

In Domain Driven Design<sup>10</sup> (DDD), a distinction is drawn between Entity Objects and Value Objects. All model objects that inherit from ActiveRecord::Base could be considered Entity Objects in DDD. An Entity object cares about identity, since each one is unique. In Active Record uniqueness is derived from the primary key. Comparing two different Entity Objects for equality should always return false, even if all of its attributes (other than the primary key) are equivalent.

Here is an example comparing two Active Record Addresses:

```
>> home = Address.create(:city => "Brooklyn", :state => "NY")
>> office = Address.create(:city => "Brooklyn", :state => "NY")
>> home == office
=> false
```

In this case you are actually creating two new Address records and persisting them to the database, therefore they have different primary key values.

Value Objects on the other hand only care that all their attributes are equal. When creating Value Objects for use with Active Record you do not inherit from

<sup>10.</sup> http://www.domaindrivendesign.org/

ActiveRecord::Base. Instead you make them part of a parent model using the composed\_of class method. This is a form of composition, called an Aggregate in DDD. The attributes of the Value Object are stored in the database together with the parent object and composed\_of provides a means to interact with those values as a single object.

A simple example is of a Person with a single Address. To model this using composition, first we need a Person model with fields for the Address. Create it with the following migration:

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
    t.string :name
    t.string :address_city
    t.string :address_state
    end
    end
end</pre>
```

The Person model looks like this:

```
class Person < ActiveRecord::Base
  composed_of :address, :mapping => [%w(address_city city),
%w(address_state state)]
end
```

We'd need a corresponding Address object which looks like this:

```
class Address
attr_reader :city, :state
def initialize(city, state)
  @city, @state = city, state
end
def ==(other_address)
  city == other_address.city && state == other_address.state
end
end
```

Note that this is just a standard Ruby object that does not inherit from ActiveRecord::Base. We have defined reader methods for our attributes and are assigning them upon initialization. We also have to define our own == method for use in comparisons. Wrapping this all up we get the following usage:

```
>> gary = Person.create(:name => "Gary")
>> gary.address_city = "Brooklyn"
>> gary.address_state = "NY"
>> gary.address
=> #<Address:0x20bc118 @state="NY", @city="Brooklyn">
```

Alternately you can instantiate the address directly and assign it using the address accessor:

```
>> gary.address = Address.new("Brooklyn", "NY")
>> gary.address
=> #<Address:0x20bc118 @state="NY", @city="Brooklyn">
```

## 9.9.1 Immutability

It's also important to treat value objects as immutable. Don't allow them to be changed after creation. Instead, create a new object instance with the new value instead. Active Record will not persist value objects that have been changed through means other than the writer method.

The immutable requirement is enforced by Active Record by freezing any object assigned as a value object. Attempting to change it afterwards will result in a ActiveSupport::FrozenObjectError.

## 9.9.2 Custom Constructors and Converters

By default value objects are initialized by calling the new constructor of the value class with each of the mapped attributes, in the order specified by the :mapping option, as arguments. If for some reason your value class does not work well with that convention, composed\_of allows a custom constructor to be specified.

When a new value object is assigned to its parent, the default assumption is that the new value is an instance of the value class. Specifying a custom converter allows the new value to be automatically converted to an instance of value class (when needed).

For example, consider the NetworkResource model with network\_address and cidr\_range attributes that should be contained in a NetAddr::CIDR value class.<sup>11</sup> The constructor for the value class is called create and it expects a CIDR address string as a parameter. New values can be assigned to the value object using either another NetAddr::CIDR object, a string or an array. The :constructor and :converter options are used to meet the requirements:

```
class NetworkResource < ActiveRecord::Base
  composed_of :cidr,
                              :class_name => 'NetAddr::CIDR',
                               :mapping => [ %w(network_address network), %w(cidr_range
bits) ],
                                 :allow_nil => true,
```

```
11. Actual objects from the NetAddr gem available at http://netaddr.rubyforge.org
```

Active Record

```
:constructor => Proc.new { |network_address, cidr_range|
NetAddr::CIDR.create("#{network address}/#{cidr range}") },
              :converter => Proc.new { |value|
NetAddr::CIDR.create(value.is a?(Array) ? value.join('/') : value) }
end
# This calls the :constructor
network_resource = NetworkResource.new(:network_address => '192.168.0.1',
:cidr_range => 24)
# These assignments will both use the :converter
network_resource.cidr = [ '192.168.2.1', 8 ]
network_resource.cidr = '192.168.0.1/24'
# This assignment won't use the :converter as the value is already an
instance of the value class
network_resource.cidr = NetAddr::CIDR.create('192.168.2.1/8')
# Saving and then reloading will use the :constructor on reload
network resource.save
```

```
network_resource.reload
```

## 9.9.3 Finding Records by a Value Object

Once a composed\_of relationship is specified for a model, records can be loaded from the database by specifying an instance of the value object in the conditions hash. The following example finds all customers with balance\_amount equal to 20 and balance\_currency equal to "USD":

```
Customer.where(:balance => Money.new(20, "USD"))
```

#### The Money Gem

A common approach to using composed\_of is in conjunction with the money gem.<sup>12</sup>

```
class Expense < ActiveRecord::Base
  composed_of :cost,
  :class_name => "Money",
  :mapping => [%w(cents cents), %w(currency currency_as_string)],
  :constructor => Proc.new do |cents, currency|
    Money.new(cents || 0, currency || Money.default_currency)
    end
end
```

Remember to add a migration with the 2 columns, the integer cents and the string currency that money needs.

<sup>12.</sup> http://github.com/FooBarWidget/money/

```
class CreateExpenses < ActiveRecord::Migration
  def self.up
    create_table :expenses do |table|
    table.integer :cents
    table.string :currency
    end
  end
  def self.down
    drop_table :expenses
  end
end
end</pre>
```

Now when asking for or setting the cost of an item would use a Money instance.

```
>> expense = Expense.create(:cost => Money.new(1000, "USD"))
>> cost = expense.cost
>> cost.cents
=> 1000
>> expense.currency
=> "USD"
```

# 9.10 Modules for Reusing Common Behavior

In this section, we'll talk about one strategy for breaking out functionality that is shared between disparate model classes. Instead of using inheritance, we'll put the shared code into modules.

In the section "Polymorphic has\_many Relationships," we described how to add a commenting feature to our recurring sample Time and Expenses application. We'll continue fleshing out that example, since it lends itself to factoring out into modules.

The requirements we'll implement are as follows: Both users and approvers should be able to add their comments to a Timesheet or ExpenseReport. Also, since comments are indicators that a timesheet or expense report requires extra scrutiny or processing time, administrators of the application should be able to easily view a list of recent comments. Human nature being what it is, administrators occasionally gloss over the comments without actually reading them, so the requirements specify that a mechanism should be provided for marking comments as "OK" first by the approver, then by the administrator.

Again, here is the polymorphic has\_many :comments, :as => :commentable that we used as the foundation for this functionality:

```
class Timesheet < ActiveRecord::Base
has_many :comments, :as => :commentable
end
```

```
class ExpenseReport < ActiveRecord::Base
has_many :comments, :as => :commentable
end
class Comment < ActiveRecord::Base
belongs_to :commentable, :polymorphic => true
end
```

Next we enable the controller and action for the administrator that list the 10 most recent comments with links to the item to which they are attached.

```
class Comment < ActiveRecord::Base
  scope :recent, order('created_at desc').limit(10)
end
class CommentsController < ApplicationController
  before_filter :require_admin, :only => :recent
  expose(:recent_comments) { Comment.recent }
end
```

Here's some of the simple view template used to display the recent comments.

```
%ul.recent.comments
- recent_comments.each do |comment|
%li.comment
%h4= comment.created_at
= comment.text
.meta
Comment on:
= link_to comment.commentable.title, comment.commentable Yes, this
would result in N+1 selects.
```

So far, so good. The polymorphic association makes it easy to access all types of comments in one listing. In order to find all of the unreviewed comments for an item, we can use a named scope on the Comment class together with the comments association.

```
class Comment < ActiveRecord::Base
  scope :unreviewed, where(:reviewed => false)
end
```

```
>> timesheet.comments.unreviewed
```

Both Timesheet and ExpenseReport currently have identical has\_many methods for comments. Essentially, they both share a common interface. They're commentable!

To minimize duplication, we could specify common interfaces that share code in Ruby by including a module in each of those classes, where the module contains the code common to all implementations of the common interface. So, mostly for the sake of
example, let's go ahead and define a Commentable module to do just that, and include it in our model classes:

```
module Commentable
has_many :comments, :as => :commentable
end
class Timesheet < ActiveRecord::Base
include Commentable
end
class ExpenseReport < ActiveRecord::Base
include Commentable
end
```

Whoops, this code doesn't work! To fix it, we need to understand an essential aspect of the way that Ruby interprets our code dealing with open classes.

# 9.10.1 A Review of Class Scope and Contexts

In many other interpreted OO programming languages, you have two phases of execution—one in which the interpreter loads the class definitions and says "this is the definition of what I have to work with," followed by the phase in which it executes the code. This makes it difficult (though not necessarily impossible) to add new methods to a class dynamically during execution.

In contrast, Ruby lets you add methods to a class at any time. In Ruby, when you type class MyClass, you're doing more than simply telling the interpreter to define a class; you're telling it to "execute the following code in the scope of this class."

Let's say you have the following Ruby script:

```
1 class Foo < ActiveRecord::Base
2 has_many :bars
3 end
4 class Foo < ActiveRecord::Base
5 belongs_to :spam
6 end
```

When the interpreter gets to line 1, you are telling it to execute the following code (up to the matching end) in the context of the Foo class object. Because the Foo class object doesn't exist yet, it goes ahead and creates the class. At line 2, we execute the statement has\_many :bars in the context of the Foo class object. Whatever the has\_many method does, it does right now.

When we again say class F00 at line 4, we are once again telling the interpreter to execute the following code in the context of the F00 class object, but this time, the

interpreter already knows about class Foo; it doesn't actually create another class. Therefore, on line 5, we are simply telling the interpreter to execute the belongs\_to :spam statement in the context of that same Foo class object.

In order to execute the has\_many and belongs\_to statements, those methods need to exist in the context in which they are executed. Because these are defined as class methods in ActiveRecord::Base, and we have previously defined class Foo as extending ActiveRecord::Base, the code will execute without a problem.

However, when we defined our Commentable module like this:

```
module Commentable
  has_many :comments, :as => :commentable
end
```

... we get an error when it tries to execute the has\_many statement. That's because the has\_many method is not defined in the context of the Commentable module object.

Given what we now know about how Ruby is interpreting the code, we now realize that what we really want is for that has\_many statement to be executed in the context of the including class.

# 9.10.2 The included Callback

Luckily, Ruby's Module class defines a handy callback that we can use to do just that. If a Module object defines the method included, it gets run whenever that module is included in another module or class. The argument passed to this method is the module/class object into which this module is being included.

We can define an included method on our Commentable module object so that it executes the has\_many statement in the context of the including class (Timesheet, ExpenseReport, and so on):

```
module Commentable
  def self.included(base)
    base.class_eval do
        has_many :comments, :as => :commentable
        end
        end
    end
end
```

Now, when we include the Commentable module in our model classes, it will execute the has\_many statement just as if we had typed it into each of those classes' bodies.

The technique is common enough, within Rails and plugins, that it was added as a first-class concept in the Rails 3 ActiveSupport API. The above example becomes shorter and easier to read as a result:

```
module Commentable
  extend ActiveSupport::Concern
  included do
    has_many :comments, :as => :commentable
  end
end
```

Whatever is inside of the included block will get executed in the class context of the class where the module is included.

has\_many :comments, :as => :commentable, :extend => Commentable

Courtenay says . . .

There's a fine balance to strike here. Magic like include Commentable certainly saves on typing and makes your model look less complex, but it can also mean that your association code is doing things you don't know about. This can lead to confusion and hours of head-scratching while you track down code in a separate module. My personal preference is to leave all associations in the model, and extend them with a module. That way you can quickly get a list of all associations just by looking at the code.

# Active Record

# 9.11 Modifying Active Record Classes at Runtime

The metaprogramming capabilities of Ruby, combined with the after\_find callback, open the door to some interesting possibilities, especially if you're willing to blur your perception of the difference between code and data. I'm talking about modifying the behavior of model classes on the fly, as they're loaded into your application.

Listing 9.5 is a drastically simplified example of the technique, which assumes the presence of a config column on your model. During the after\_find callback, we get a handle to the unique singleton class<sup>13</sup> of the model instance being loaded. Then we execute the contents of the config attribute belonging to this particular Account instance, using Ruby's class\_eval method. Since we're doing this using the singleton class for this instance, rather than the global Account class, other account instances in the system are completely unaffected.

<sup>13.</sup> I don't expect this to make sense to you, unless you are familiar with Ruby's singleton classes, and the ability to evaluate arbitrary strings of Ruby code at runtime. A good place to start is http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html.

Listing 9.5 Runtime metaprogramming with after\_find

```
class Account < ActiveRecord::Base
   ...
   protected
   def after_find
    singleton = class << self; self; end
    singleton.class_eval(config)
   end
end</pre>
```

I used powerful techniques like this one in a supply-chain application that I wrote for a large industrial client. A lot is a generic term in the industry used to describe a shipment of product. Depending on the vendor and product involved, the attributes and business logic for a given lot vary quite a bit. Since the set of vendors and products being handled changed on a weekly (sometimes daily) basis, the system needed to be reconfigurable without requiring a production deployment.

Without getting into too much detail, the application allowed the maintenance programmers to easily customize the behavior of the system by manipulating Ruby code stored in the database, associated with whatever product the lot contained.

For example, one of the business rules associated with lots of butter being shipped for Acme Dairy Co. might dictate a strictly integral product code, exactly 10 digits in length. The code, stored in the database, associated with the product entry for Acme Dairy's butter product would therefore contain the following two lines:

```
validates_numericality_of :product_code, :only_integer => true
validates_length_of :product_code, :is => 10
```

## 9.11.1 Considerations

A relatively complete description of everything you can do with Ruby metaprogramming, and how to do it correctly, would fill its own book. For instance, you might realize that doing things like executing arbitrary Ruby code straight out of the database is inherently dangerous. That's why I emphasize again that the examples shown here are very simplified. All I want to do is give you a taste of the possibilities.

If you do decide to begin leveraging these kinds of techniques in real-world applications, you'll have to consider security and approval workflow and a host of other important concerns. Instead of allowing arbitrary Ruby code to be executed, you might feel compelled to limit it to a small subset related to the problem at hand. You might design a compact API, or even delve into authoring a domain-specific language (DSL), crafted specifically for expressing the business rules and behaviors that should be loaded dynamically. Proceeding down the rabbit hole, you might write custom parsers for your DSL that could execute it in different contexts—some for error detection and others for reporting. It's one of those areas where the possibilities are quite limitless.

## 9.11.2 Ruby and Domain-Specific Languages

My former colleague Jay Fields and I pioneered the mix of Ruby metaprogramming, Rails, and internal<sup>14</sup> domain-specific languages while doing Rails application development for clients. I still occasionally speak at conferences and blog about writing DSLs in Ruby.

Jay has also written and delivered talks about his evolution of Ruby DSL techniques, which he calls Business Natural Languages (or BNL for short<sup>15</sup>). When developing BNLs, you craft a domain-specific language that is not necessarily valid Ruby syntax, but is close enough to be transformed easily into Ruby and executed at runtime, as shown in Listing 9.6.

Active Record

Listing 9.6 Example of business natural language

employee John Doe compensate 500 dollars for each deal closed in the past 30 days compensate 100 dollars for each active deal that closed more than 365 days ago compensate 5 percent of gross profits if gross profits are greater than 1,000,000 dollars compensate 3 percent of gross profits if gross profits are greater than 2,000,000 dollars compensate 1 percent of gross profits if gross profits are greater than 3,000,000 dollars

The ability to leverage advanced techniques such as DSLs is yet another powerful tool in the hands of experienced Rails developers.

The qualifier internal is used to differentiate a domain-specific language hosted entirely inside of a generalpurpose language, such as Ruby, from one that is completely custom and requires its own parser implementation.
 Googling BNL will give you tons of links to the Toronto-based band Barenaked Ladies, so you're better off going directly to the source at http://bnl.jayfields.com.

# 9.12 Conclusion

With this chapter we conclude our coverage of Active Record. Among other things, we examined how callbacks and observers let us factor our code in a clean and objectoriented fashion. We also expanded our modeling options by considering single-table inheritance, abstract classes and Active Record's distinctive polymorphic relationships.

At this point in the book, we've covered two parts of the MVC pattern: the model and the controller. It's now time to delve into the third and final part: the view.

#### Courtenay says ...

DSLs suck! Except the ones written by Obie, of course. The only people who can read and write most DSLs are their original authors. As a developer taking over a project, it's often quicker to just reimplement instead of learning the quirks and exactly which words you're allowed to use in an existing DSL.In fact, a lot of Ruby metaprogramming sucks, too. It's common for people gifted with these new tools to go a bit overboard. I consider metaprogramming, self.included, class\_eval, and friends to be a bit of a code smell on most projects. If you're making a web application, future developers and maintainers of the project will appreciate your using simple, direct, granular, and well-tested methods, rather than monkeypatching into existing classes, or hiding associations in modules. That said, if you can pull it off ... your code will become more powerful than you can possibly imagine.

# A

Action Controller communication with view, 104-105 controller specs, 528-531 filters, 105-111 around, 109 classes 107 conditions, 110 halting chain, 111 inheritance, 106 ordering, 108 skipping, 110 layouts, specifying, 101 post-backs, 356 rendering, 92-101 standard RESTful actions, 61-64 streaming content, 112-116 verify method, 111–112 Action Dispatch, 88-91 Action Mailer, 473-483 attachments, 475, 477 custom email headers, 475 generating URLs inside messages, 478 handing inbound attachments, 480 HTML messages, 476

mailer layouts, 478 models, 474 multipart messages, 477 preparing outbound messages, 474 raising delivery errors, 19 receiving, 479-481 sending, 479 server configuration, 481 SMTP, 473 testing with RSpec, 481-483 Action View, 293-308 conditional output, 296 customizing validation error output, 315 ERb, 293 filename conventions, 294 flash messages, 300-302 Haml, 293 instance variables, 297-302 layouts, 294-295 partials, see Partials. view specs, 531-533 yielding content, 295 Active Model, 563-580 AttributeMethods module, 563-564 Callbacks module, 565

Conversion module, 565-566 Dirty module, 566 Errors class, 567 MassAssignmentSecurity module, 569 Naming module, 571 observers, 571-573 serialization, 573-575 testing compatibility of custom classes with Lint:: Tests, 569 translation, 575 Validations module, 576-580 Active Resource 459-471 authentication, 467 customizing default URLs, 465 Active Record abstract base models, 276 associations, 121, 181-230 :counter\_cache option, 190, 195 :counter\_sql option, 187 :dependent option, 187, 188, 195 :finder\_sql option, 187 AssociationProxy class, 228-229 belongs\_to. See belongs\_to associations checking inclusion of records in collection, 189 class hierarchy, 181 destroying records, 188 extensions, 226-227 foreign-key constraints, 281 has\_and\_belongs\_to\_many. See has\_and\_ belongs\_to\_many associations has\_many :through. See has\_many :through associations has\_many. See has\_many associations indexing, 484 many-to-many relationships, 208-214 one-to-many relationships, 183-190 one-to-one relationships, 222-225 polymorphic, 277-281

size of, 190 unique sets, 190 unsaved objects, 225 attributes, 123-126 controlling access, 140 readonly, 141 reloading, 131 serialized, 125 typecasting, 131 updating, 136-1140 Base class, 120 basic object operations, 127-133 calculation methods, 265-267 callbacks, 256-265 list of, 258-259 cloning, 131 concurrency. See Database locking configuration, 158 dynamic finder methods, 132 dynamic scopes, 133 find\_by\_sql method, 133-134 legacy naming schemes, 122-123 model specs, 526-528 migrations, 161-179 column type mappings, 168-172 creating, 161-172 magic timestamp columns, 172 schema.rb file, 174 sequencing, 162 observers, 10, 268-269 pattern, 119 query caching, 135-136 querying, 146-152 exists, 152 from, 150 group, 150 having, 150 includes, 151 joins, 151

#### 688

limit, 149 offset, 149 order, 148 readonly, 152 select, 149 where, 146-148 RecordInvalid exception, 187 RecordNotSaved exception, 187, 188 records deleting, 141-142 random ordering, 148 touching, 139 scopes, 251-255 session store, 429 STI (Single-Table Inheritance), 269-276 translations, 386-388, 390 validations, 231-250 common options, 242-243 conditional validation, 243-245 contexts, 245 custom macros, 247-248 errors, 231-232, 249-250 enforcing uniqueness of join models, 240 reporting, 310-312 short-form, 245-246 skipping, 249 testing with Shoulda, 250 value objects, 281-285 Active Support, 581-688 Ajax, 409-425 changes in Rails 3, 410 CSS selectors, 418 HTML fragments, 421 JSON, 419-421 JSONP, 423-424 Unobtrusive JavaScript (UJS), 411-412 Array, extensions, 581-587 Asset hosts, 22, 321-323 Asset timestamps, 323

Asynchronous processing, See Background processing. Atom Feeds autodetection, 316–317 atom\_feed method, 324–326 Authentication, 435 Active Resource, 467 client-side certificates, 468 HTTP basic, 467 HTTP digest, 468 Authlogic, 436 configuration, 439–440

## B

background processing, 551–561 Base64 class, 588 BasicObject class, 588-589 belongs\_to associations, 191-199 building and creating related objects, 192 options, 192-199 polymorphic, 197 reloading, 191 touch, 198 with conditions, 193 benchmarking, 589 binary data storage, 170 breadcrumbs, 400-401 Builder::XmlMarkup class, 456 Bundler, 2-7 loading gems directly from Git repository, 4 - 5

# С

Caching :counter\_cache, 190, 195 action caching, 486–487 CacheHelper module, 326 controlling web caches and proxies, 499 disabling in development mode, 18 ETags, 500–502

expiration, 490-493 fetch, 498-499 fragment caching, 488-490 page caching, 486 query caching, 135-136 storage, 495 Store class, 592-597 sweeping, 493, 496 view caching, 485-497 Callbacks. See also Active Record, callbacks in Active Support, 597-599 CAS, 445 CDATA, 366 chars proxy, 647-650 Class, extensions, 600 Concern module, 604 Concurrency. See Database Locking Configurable module, 605 const\_missing method, 646 Controllers. See Action Controller convention over configuration, 119, 122 Cookies :secure option, 434 integrity, 11-12 reading and writing, 433 session store, 431 signing, 434 CRUD (Create Read Update Delete), 119 CSS linking stylesheets to template, 318-319 sanitizing, 365 Currency formatting, 359 Money gem, 284

#### D

data migration, 173–174 Databases connecting to multiple, 153–154 foreign-key constraints, 281

locking, 142-146 considerations, 145 optimistic, 143 pessimistic, 145 migrations. See ActiveRecord, Migrations. schemas, 15, 161 seeding, 175-76 using directly, 154-158 Date, extensions, 605-611 Date input tags, 328-331 DateTime, extensions, 611 Decent Exposure gem, 105, 297-298 decimal precision, 169, 171 Delayed Job gem, 552-555 Deprecation, 619 Devise gem, 441 Domain-Specific Languages, 291 Drag and Drop, 417 Duration class, 619-620

#### E

Email. See Action Mailer Enumerable, extensions, 621–622 ETags, 500–502 Excerpting text, 370

## F

Facebook, 445 favicon.ico file, 317 Files extensions by Active Support, 623–624 reporting sizes to users, 359 upload field, 348, 356 Firebug, 410 floats, 171 Forms \_destroy checkbox, 345 \_method hidden field, 64 accepts\_nested\_attributes\_for method, \_344–345

attributes not typecasted, 343 automatic view creation, 313–314 button\_to helper method, 391 custom builder classes, 347 dynamically adding rows of child records, 338 helper methods, 333–358 input, 348–358 updating multiple objects at once, 337

## G

Gemfile, 3 Geocoding, 260–261

#### Η

has\_and\_belongs\_to\_many associations, 208-214 bidirectional, 210 custom SQL, 211-213 extra columns, 213 making self-referential, 209 has\_many :through associations, 214-221 and validations, 218 join models, 215 options, 219-221 usage, 216 has\_many associations, 199-208 :class\_name option, 202 :conditions option, 202 :include option, 204-206 callbacks, 200-201 has\_one associations, 222-225 options, 224-225 together with has\_many, 223 Hash, extensions, 624-629 HashWithIndifferentAccess class, 629 Heckle, 534 Helper methods, helper specs, 533 writing your own 398-407

escaping, 367 sanitizing, 364-365 tags a. 392-394 audio, 319 label, 348 form. See Forms. option, 353-355 password, 349 select, 350-351 script, 359 submit, 349 video, 320 HTTP basic authentication, 467 foundation of REST, 55-56 stateless, 427 status codes, 99-101 verbs (GET, POST, etc.), 60-64, 393

## I

HTML

IMAP, 445 Image tags, 320 Initializers, 11-14 backtrace\_silencers.rb, 11 cookie\_verification\_secret.rb, 11-12 inflections.rb, 12-13 mime\_types.rb, 14 session\_store.rb, 14, 430 Inflections. See Pluralization Integer, extensions, 634-635 Internationalization (I18n), 372-391 Active Model, 575 Active Record, 386 default locale, 10 exception handling, 391 interpolation, 385 locale files, 382-383 process, 380-390

setting user locales, 377–380 setup, 374–380

# J

JavaScript, 97, 317–318, 358–259, 409–425 escaping, 358 including in template, 317–318 link\_to method enhancements, 392–393 using to insert HTML into pages, 338 JQuery framework, 410–411, 418, 421, 423–424 JSON, 97, 419–421, 635–636 JSONP, 423–424

## K

Kernel, extensions, 636-637

#### L

LDAP, 445 link\_to helper methods, 392–394 Locale files, 382–383 Logging, 23–28 backtrace silencing, 11, 587 BufferedLogger class, 590–592 colorization, 27 level override, 15 levels, 23–24 log file analysis, 26–27 Logger, extensions, 637–638 Syslog, 28

#### M

Memcache, session store, 430 MessageEncryptor class, 638–639 MessageVerifier class, 639–640 Middleware (Rack), 86–88 MIME types, 13–14 Module, extensions, 640 MongoDB, 444 MVC (Model-View-Controller), 85

#### N

Named scopes. See Active Record, scopes. Nonces, 432 Notifications, 651–652 Numbers delimiters, 360 extensions to Numeric class, 652–655 conversion, 359–361

## 0

Object, extensions, 655 Observers, 10 OpenID, 445 OpenSSL Digests, 431

#### Р

params hash, 336-338 Partials, 95, 302-307 passing variables to, 305 rendering collections, 306 reuse, 303 shared, 304 wrapping and generalizing, 401-407 Plugins, 537-550 as RubyGems, 538 extending Rails classes, 542 installation and removal, 544-545 load order, 9-10 plugin script, 538 rake tasks, 545-546 testing, 547-548 writing your own, 539-550 Pluralization i18n, 385 Inflector class, 12-13 Inflections class, 630-634 pluralize helper method, 370 Prototype framework, 361, 411 helper methods, 361 Prototype Legacy Helper plugin, 413

## R

Rack, 86-88, 90-91 Rack::Sendfile middleware, 114 RACK\_ENV variable, 1 routes as Rack endpoints, 41-42 rails.js file, 411 Rails Class loader and reloading, 16-18, 615-619 console, 12 reloading, 92 Engines, 549 lib directory, 17 root directory, 9 runner, 559 scaffolding, 311-314 settings, 1-29 application.rb file, 8-11 autoload paths, 9 boot.rb file, 8 cherry-picking frameworks used, 8 custom environments, 20 development mode, 15-19 environment.rb file, 8 generator defaults, 11 initializers. See Initializers production mode, 20-23 test mode, 19-20 RAILS\_ENV variable, 1 Railties, 548-549, 660 Rake tasks (selected), db:migrate, 163, 176 log:clear 24 routes, 53 spec, 523 Random ordering of records, 148 SecureRandom generator class, 663-664 Range, extensions, 660-661 RecordNotFound exception, 128

Regexp, extensions, 662 Rendering views, 92-101 another actions's template, 93 explicit, 93, 94 implicit, 92-93 inline templates, 96 **ISON**, 97 nothing, 97 options, 98 partials, see Partials. text, 96 XML, 97 Request handling in routing, 89 redirecting, 101-104, 418 verification, 111-112 Rescuable module, 662 Resque gem, 555-559 REST and RESTful design, 55-83 action set, 78-82 collection routes, 72 controller-only resources, 74 forms, 335 member routes, 70-71 HTTP verbs, 60-64 nested resources, 65-69 routes, 31, 58-61 resources and representations, 40-41, 56-57, 76-77 singular resource routes, 64-65 standard controller actions, 61-64 REXML, 458, 686 Roy T. Fielding. See also REST and RESTful design, 55-58 Routing, 31–54 constraining by request method, 38-39 formats, 40 globbing, 45-46 listing, 53

match method, 34-37 named, 46-50 RESTful routes, 31, 58-61 :format parameter, 76 collection, 72 member, 70-71 nested, 65-69 singular, 64-65 redirecting, 39-40 root routes, 44-45 routes.rb file, 33-34 scopes, 50-53 RJS, 412-419 templates, 413 RSS autodetection, 316-317 RPX authentication, 445 RSpactor, 533 RSpec, 503-535 assertions, 512 custom expectation matchers, 516 generator settings, 11 grouping related examples, 506 let methods, 506-508 mocking and stubbing, 519-522 pending, 511-512 predicate matchers, 515-516 running specs, 522 runtime options, 524 shared behaviors, 519 spec\_helper.rb file, 517, 524-526 subjects, 513-515 testing email, 481-483 Ruby \$LOAD\_PATH, 9, 16 hashes, 452 macro-style methods, 121 Marshal API, 427 modules for reusing common behavior, 285-289

RubyGems as plugins, 538 Bundler, 2–7 Git repository, loading directly from, 4 installing, 5–7 packaging, 7 using pre-release gems, 4

## S

Scopes, see Active Record, scopes Security CSRF attacks, 336 replay attacks, 431 SQL injection, 134 Session Management, 427-434 cleaning old sessions, 432 storage **RESTful considerations**, 75 turning off sessions, 429 Settings, 1-29 Specjour, 534 Spork, 534 SSL OpenSSL digests, 431 serving protected assets, 323 X.509 certificates, 467-468 Static content, 116 Streaming, 112-116 String, extensions, 664-673 usage versus symbols, 130 StringInquirer class, 673 SOAP, 459 Symbol, extensions, 673 usage versus strings, 130

#### Т

Templates. See View templates. Thread safety, 22–23

Time extensions, 675–682 input tags, 328–331 reporting distances in time, 332–333 storing in database, 170 Time Zones DateTime conversions, 614 default, 10 option tags helper, 354–355 TimeZone class, 683 TimeWithZone class, 682 Truncating text, 372

## U

Unicode, 364, 385, 648–649 Unobtrusive JavaScript (UJS), 411–412 URL generation, 395–398 patterns in routing, 35–36 segment keys, 36–38

## V

Validation. See Active Record, validations Value objects, 281–285 View templates. See also Action View, 293–308 capturing block content, 326–327 concat method, 368 cycling content, 369 debugging output, 333 encapsulating logic in helper methods, 399 highlighting content, 370 localization, 373 raw output, 361 record identification, 362–364 transforming text into HTML, 371 translation. See Internationalization. word wrap, 372 Visual effects, 419

## W

Watchr, 534 Web 2.0, 332, 425 Web architecture, 55–56 Whiny nils, 18

## X

XML, 447–459 parsing, 458 to\_xml method, 447–456 Active Record associations, 450 customizing output, 448–450 extra elements, 454 overriding, 455–457 Ruby hashes, 452 typecasting, 459 XML Builder, 456–458 XMLHttpRequestObject, 409 XMLMini module, 686–688

#### Y

YAML, 125-126, 447

This page intentionally left blank