

Addison-Wesley Professional Ruby Series



THE RAILS™ 4 WAY

OBIE FERNANDEZ
KEVIN FAUSTINO

Foreword by STEVE KLABNIK

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Praise for *The Rails Way*

For intermediates and above, I strongly recommend adding this title to your technical bookshelf. There is simply no other Rails title on the market at this time that offers the technical depth of the framework than *The Rails™ 3 Way*.

—Mike Riley, *Dr. Dobb's Journal*

I highly suggest you get this book. Software moves fast, especially the Rails API, but I feel this book has many core API and development concepts that will be useful for a while to come.

—Matt Polito, software engineer and member of Chicago Ruby User Group

This book should live on your desktop if you're a Rails developer. It's nearly perfect in my opinion.

—Luca Pette, developer

The Rails™ 3 Way is likely to take you from being a haphazard poke-a-stick-at-it programmer to a deliberate, skillful, productive, and confident RoR developer.

—Katrina Owen, JavaRanch

I can positively say that it's the single best Rails book ever published to date. By a long shot.

—Antonio Cangiano, software engineer and technical evangelist at IBM

This book is a great crash course in Ruby on Rails! It doesn't just document the features of Rails, it filters everything through the lens of an experienced Rails developer—so you come out a pro on the other side.

—Dirk Elmendorf, cofounder of Rackspace Inc. and Rails developer

The key to *The Rails Way* is in the title. It literally covers the “way” to do almost everything with Rails. Writing a truly exhaustive reference to the most popular web application framework used by thousands of developers is no mean feat. A thankful community of developers that has struggled to rely on scant documentation will embrace *The Rails Way* with open arms. A tour de force!

—Peter Cooper, editor, *Ruby Inside: The Ruby Blog*

In the past year, dozens of Rails books have been rushed to publication. A handful are good. Most regurgitate rudimentary information easily found on the Web. Only this book provides both the broad and deep technicalities of Rails. Nascent and expert developers, I recommend you follow *The Rails Way*.

—Martin Streicher, chief technology officer, McClatchy Interactive, former editor in chief of *Linux Magazine*

Hal Fulton's *The Ruby Way* has always been by my side as a reference while programming Ruby. Many times I had wished there was a book that had the same depth and attention to detail, only focused on the Rails framework. That book is now here and hasn't left my desk for the past month.

—Nate Klaiber, Ruby programmer

I knew soon after becoming involved with Rails that I had found something great. Now, with Obie's book, I have been able to step into Ruby on Rails development coming from .NET and be productive right away. The applications I have created I believe to be a much better quality due to the techniques I learned using Obie's knowledge.

—Robert Bazinet, InfoQ.com, .NET, and Ruby community editor and founding member of the Hartford Ruby Brigade

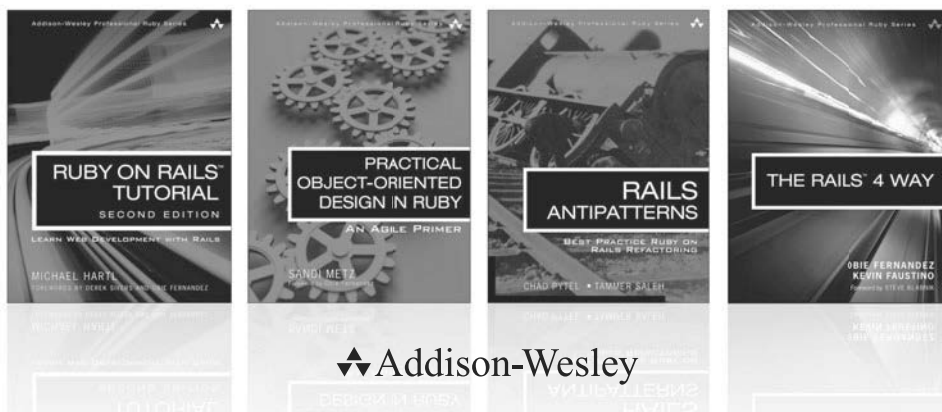
Extremely well written; it's a resource that every Rails programmer should have. Yes, it's that good.

—Reuven Lerner, *Linux Journal* columnist

THE RAILS™ 4 WAY

Addison-Wesley Professional Ruby Series

Obie Fernandez, Series Editor



Visit informit.com/ruby for a complete list of available products.

The Addison-Wesley Professional Ruby Series provides readers with practical, people-oriented, and in-depth information about applying the Ruby platform to create dynamic technology solutions. The series is based on the premise that the need for expert reference books, written by experienced practitioners, will never be satisfied solely by blogs and the Internet.



Make sure to connect with us!
informit.com/socialconnect

informit.com
the trusted technology learning source

Addison-Wesley

Safari
Books Online

THE RAILS™ 4 WAY

Obie Fernandez

Kevin Faustino

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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 authors 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.

For information about buying this title in bulk quantities or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the web: informit.com/aw

Cataloging-in-Publication Data is on file with the Library of Congress

Copyright © 2014 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

Parts of this book contain material excerpted from the Ruby and Rails source code and API documentation, copyright © 2004–2014 by David Heinemeier Hansson under the MIT license. Chapter 21 contains material excerpted from the RSpec source code and API documentation, copyright © 2005–2014 by the RSpec Development Team. Rails and Ruby on Rails are trademarks of David Heinemeier Hansson.

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 WARRANTIES 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: 978-0-321-94427-6

ISBN-10: 0-321-94427-5

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, June 2014

Editor-in-Chief
Mark L. Taub

Acquisitions Editor
Debra Williams Cauley

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Editorial Assistant
Kim Boedigheimer

Production Editor
Scribe Inc.

Copy Editor
Scribe Inc.

Indexer
Scribe Inc.

Proofreader
Scribe Inc.

Compositor
Scribe Inc.

Cover Designer
Chuti Prasertsith

*Taylor, your hard work and dedication to your craft are an inspiration
to me every day. I love you.*

This page intentionally left blank

Contents

Foreword by Steve Klabnik	xxxix
Foreword to the Previous Edition by David Heinemeier Hansson	xli
Foreword to the Previous Edition by Yehuda Katz	xlili
Introduction	xlvi
Acknowledgments	li
About the Authors	liii

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	9
1.2.1	<code>config/application.rb</code>	9
1.2.2	Initializers	11
1.2.3	Additional Configuration	16
1.2.4	Spring Application Preloader	17
1.3	Development Mode	18
1.3.1	Automatic Class Reloading	18
1.3.2	Eager Load	20
1.3.3	Error Reports	20
1.3.4	Caching	20
1.3.5	Raise Delivery Errors	21

1.3.6	Deprecation Notices	21
1.3.7	Pending Migrations Error Page	21
1.3.8	Assets Debug Mode	22
1.4	Test Mode	22
1.5	Production Mode	23
1.5.1	Assets	25
1.5.2	Asset Hosts	26
1.6	Configuring a Database	26
1.7	Configuring Application Secrets	27
1.8	Logging	29
1.8.1	Rails Log Files	30
1.8.2	Tagged Logging	32
1.8.3	Log File Analysis	32
1.9	Conclusion	35

Chapter 2 Routing 37

2.1	The Two Purposes of Routing	38
2.2	The <code>routes.rb</code> File	39
2.2.1	Regular Routes	40
2.2.2	Constraining Request Methods	41
2.2.3	URL Patterns	42
2.2.4	Segment Keys	43
2.2.5	Spotlight on the <code>:id</code> Field	44
2.2.6	Optional Segment Keys	44
2.2.7	Redirect Routes	45
2.2.8	The Format Segment	46
2.2.9	Routes as Rack Endpoints	48
2.2.10	Accept Header	48
2.2.11	Segment Key Constraints	49
2.2.12	The Root Route	50
2.3	Route Globbing	51
2.4	Named Routes	53
2.4.1	Creating a Named Route	53
2.4.2	<code>name_path</code> versus <code>name_url</code>	54
2.4.3	What to Name Your Routes	55
2.4.4	Argument Sugar	56
2.4.5	A Little More Sugar with Your Sugar?	56

- 2.5 Scoping Routing Rules 57
 - 2.5.1 Controller 58
 - 2.5.2 Path Prefix 58
 - 2.5.3 Name Prefix 58
 - 2.5.4 Namespaces 59
 - 2.5.5 Bundling Constraints 59
- 2.6 Listing Routes 60
- 2.7 Conclusion 61

Chapter 3 REST, Resources, and Rails 63

- 3.1 REST in a Rather Small Nutshell 63
- 3.2 Resources and Representations 64
- 3.3 REST in Rails 65
- 3.4 Routing and CRUD 66
 - 3.4.1 REST Resources and Rails 66
 - 3.4.2 From Named Routes to REST Support 67
 - 3.4.3 Reenter the HTTP Verb 68
- 3.5 The Standard RESTful Controller Actions 69
 - 3.5.1 PATCH versus PUT 70
 - 3.5.2 Singular and Plural RESTful Routes 71
 - 3.5.3 The Special Pairs: `new/create` and `edit/update` 71
 - 3.5.4 The PATCH and DELETE Cheat 72
 - 3.5.5 Limiting Routes Generated 73
- 3.6 Singular Resource Routes 73
- 3.7 Nested Resources 74
 - 3.7.1 RESTful Controller Mappings 75
 - 3.7.2 Considerations 75
 - 3.7.3 Deep Nesting? 76
 - 3.7.4 Shallow Routes 77
- 3.8 Routing Concerns 78
- 3.9 RESTful Route Customizations 79
 - 3.9.1 Extra Member Routes 80
 - 3.9.2 Extra Collection Routes 81
 - 3.9.3 Custom Action Names 82
 - 3.9.4 Mapping to a Different Controller 82
 - 3.9.5 Routes for New Resources 82
 - 3.9.6 Considerations for Extra Routes 83

- 3.10 Controller-Only Resources 83
- 3.11 Different Representations of Resources 86
 - 3.11.1 The `respond_to` Method 86
 - 3.11.2 Formatted Named Routes 87
- 3.12 The RESTful Rails Action Set 88
 - 3.12.1 Index 88
 - 3.12.2 Show 90
 - 3.12.3 Destroy 90
 - 3.12.4 New and Create 91
 - 3.12.5 Edit and Update 92
- 3.13 Conclusion 92

Chapter 4 Working with Controllers 95

- 4.1 Rack 96
 - 4.1.1 Configuring Your Middleware Stack 97
- 4.2 Action Dispatch: Where It All Begins 99
 - 4.2.1 Request Handling 99
 - 4.2.2 Getting Intimate with the Dispatcher 99
- 4.3 Render unto View... 102
 - 4.3.1 When in Doubt, Render 102
 - 4.3.2 Explicit Rendering 103
 - 4.3.3 Rendering Another Action's Template 103
 - 4.3.4 Rendering a Different Template Altogether 104
 - 4.3.5 Rendering a Partial Template 105
 - 4.3.6 Rendering Inline Template Code 106
 - 4.3.7 Rendering Text 106
 - 4.3.8 Rendering Other Types of Structured Data 107
 - 4.3.9 Rendering Nothing 108
 - 4.3.10 Rendering Options 108
- 4.4 Additional Layout Options 111
- 4.5 Redirecting 111
 - 4.5.1 The `redirect_to` Method 113
- 4.6 Controller/View Communication 115
- 4.7 Action Callbacks 116
 - 4.7.1 Action Callback Inheritance 116
 - 4.7.2 Action Callback Types 117
 - 4.7.3 Action Callback Chain Ordering 118
 - 4.7.4 Around Action Callbacks 119
 - 4.7.5 Action Callback Chain Skipping 120

- 4.7.6 Action Callback Conditions 120
- 4.7.7 Action Callback Chain Halting 121
- 4.8 Streaming 121
 - 4.8.1 ActionController::Live 121
 - 4.8.2 View Streaming via `render stream: true` 122
 - 4.8.3 `send_data(data, options = {})` 123
 - 4.8.4 `send_file(path, options = {})` 124
- 4.9 Variants 126
- 4.10 Conclusion 127

Chapter 5 Working with Active Record 129

- 5.1 The Basics 130
- 5.2 Macro-Style Methods 131
 - 5.2.1 Relationship Declarations 131
 - 5.2.2 Convention over Configuration 132
 - 5.2.3 Setting Names Manually 132
 - 5.2.4 Legacy Naming Schemes 133
- 5.3 Defining Attributes 133
 - 5.3.1 Default Attribute Values 134
 - 5.3.2 Serialized Attributes 136
 - 5.3.3 ActiveRecord::Store 137
- 5.4 CRUD: Create, Read, Update, and Delete 138
 - 5.4.1 Creating New Active Record Instances 138
 - 5.4.2 Reading Active Record Objects 139
 - 5.4.3 Reading and Writing Attributes 139
 - 5.4.4 Accessing and Manipulating Attributes before They Are Typecast 142
 - 5.4.5 Reloading 142
 - 5.4.6 Cloning 142
 - 5.4.7 Custom SQL Queries 143
 - 5.4.8 The Query Cache 144
 - 5.4.9 Updating 145
 - 5.4.10 Updating by Condition 147
 - 5.4.11 Updating a Particular Instance 147
 - 5.4.12 Updating Specific Attributes 148
 - 5.4.13 Convenience Updaters 149
 - 5.4.14 Touching Records 149
 - 5.4.15 `readonly` Attributes 149
 - 5.4.16 Deleting and Destroying 150

5.5	Database Locking	151
5.5.1	Optimistic Locking	152
5.5.2	Pessimistic Locking	154
5.5.3	Considerations	154
5.6	Where Clauses	155
5.6.1	where(*conditions)	155
5.6.2	order(*clauses)	157
5.6.3	limit(number) and offset(number)	158
5.6.4	select(*clauses)	159
5.6.5	from(*tables)	159
5.6.6	exists?	160
5.6.7	extending(*modules, &block)	160
5.6.8	group(*args)	160
5.6.9	having(*clauses)	161
5.6.10	includes(*associations)	161
5.6.11	joins	162
5.6.12	none	162
5.6.13	readonly	163
5.6.14	references	163
5.6.15	reorder	163
5.6.16	reverse_order	164
5.6.17	uniq / distinct	164
5.6.18	unscope(*args)	164
5.6.19	arel_table	165
5.7	Connections to Multiple Databases in Different Models	165
5.8	Using the Database Connection Directly	167
5.8.1	The DatabaseStatements Module	167
5.8.2	Other Connection Methods	169
5.9	Other Configuration Options	171
5.10	Conclusion	171

Chapter 6 Active Record Migrations 173

6.1	Creating Migrations	173
6.1.1	Sequencing Migrations	174
6.1.2	change	174
6.1.3	reversible	175
6.1.4	Irreversible Migrations	176

- 6.1.5 `create_table(name, options, &block)` 176
- 6.1.6 `change_table(table_name, &block)` 178
- 6.1.7 `create_join_table` 178
- 6.1.8 API Reference 178
- 6.1.9 Defining Columns 181
- 6.1.10 Command-Line Column Declarations 187
- 6.2 Data Migration 187
 - 6.2.1 Using SQL 187
 - 6.2.2 Migration Models 188
- 6.3 `schema.rb` 189
- 6.4 Database Seeding 190
- 6.5 Database-Related Rake Tasks 191
 - 6.5.1 `db:migrate:status` 193
- 6.6 Conclusion 194

Chapter 7 Active Record Associations 195

- 7.1 The Association Hierarchy 195
- 7.2 One-to-Many Relationships 196
 - 7.2.1 Adding Associated Objects to a Collection 198
 - 7.2.2 Association Collection Methods 199
- 7.3 The `belongs_to` Association 205
 - 7.3.1 Reloading the Association 205
 - 7.3.2 Building and Creating Related Objects via the Association 206
 - 7.3.3 `belongs_to` Options 206
 - 7.3.4 `belongs_to` Scopes 211
- 7.4 The `has_many` Association 214
 - 7.4.1 `has_many` Options 214
 - 7.4.2 `has_many` Scopes 218
- 7.5 Many-to-Many Relationships 222
 - 7.5.1 `has_and_belongs_to_many` 222
 - 7.5.2 `has_many :through` 226
 - 7.5.3 `has_many :through` Options 230
 - 7.5.4 Unique Association Objects 232
- 7.6 One-to-One Relationships 233
 - 7.6.1 `has_one` 233
 - 7.6.2 `has_one` Scopes 236

- 7.7 Working with Unsaved Objects and Associations 236
 - 7.7.1 One-to-One Associations 236
 - 7.7.2 Collections 237
 - 7.7.3 Deletion 237
- 7.8 Association Extensions 238
- 7.9 The CollectionProxy Class 239
- 7.10 Conclusion 240

Chapter 8 Validations 241

- 8.1 Finding Errors 241
- 8.2 The Simple Declarative Validations 242
 - 8.2.1 validates_absence_of 242
 - 8.2.2 validates_acceptance_of 242
 - 8.2.3 validates_associated 243
 - 8.2.4 validates_confirmation_of 244
 - 8.2.5 validates_each 244
 - 8.2.6 validates_format_of 245
 - 8.2.7 validates_inclusion_of and
validates_exclusion_of 246
 - 8.2.8 validates_length_of 246
 - 8.2.9 validates_numericality_of 247
 - 8.2.10 validates_presence_of 248
 - 8.2.11 validates_uniqueness_of 249
 - 8.2.12 validates_with 251
 - 8.2.13 RecordInvalid 252
- 8.3 Common Validation Options 253
 - 8.3.1 :allow_blank and :allow_nil 253
 - 8.3.2 :if and :unless 253
 - 8.3.3 :message 253
 - 8.3.4 :on 254
 - 8.3.5 :strict 254
- 8.4 Conditional Validation 255
 - 8.4.1 Usage and Considerations 255
 - 8.4.2 Validation Contexts 256
- 8.5 Short-Form Validation 256
- 8.6 Custom Validation Techniques 258
 - 8.6.1 Add Custom Validation Macros to Your
Application 258

- 8.6.2 Create a Custom Validator Class 259
- 8.6.3 Add a `validate` Method to Your Model 260
- 8.7 Skipping Validations 260
- 8.8 Working with the Errors Hash 261
 - 8.8.1 Checking for Errors 261
- 8.9 Testing Validations with `Shoulda` 262
- 8.10 Conclusion 262

Chapter 9 Advanced Active Record 263

- 9.1 Scopes 263
 - 9.1.1 Scope Parameters 264
 - 9.1.2 Chaining Scopes 264
 - 9.1.3 Scopes and `has_many` 265
 - 9.1.4 Scopes and Joins 265
 - 9.1.5 Scope Combinations 265
 - 9.1.6 Default Scopes 266
 - 9.1.7 Using Scopes for CRUD 267
- 9.2 Callbacks 268
 - 9.2.1 One-Liners 269
 - 9.2.2 Protected or Private 269
 - 9.2.3 Matched `before/after` Callbacks 269
 - 9.2.4 Halting Execution 271
 - 9.2.5 Callback Usages 271
 - 9.2.6 Special Callbacks: `after_initialize` and `after_find` 274
 - 9.2.7 Callback Classes 276
- 9.3 Calculation Methods 278
 - 9.3.1 `average(column_name, *options)` 279
 - 9.3.2 `count(column_name, *options)` 279
 - 9.3.3 `ids` 279
 - 9.3.4 `maximum(column_name, *options)` 279
 - 9.3.5 `minimum(column_name, *options)` 279
 - 9.3.6 `pluck(*column_names)` 279
 - 9.3.7 `sum(column_name, *options)` 279
- 9.4 Single-Table Inheritance (STI) 280
 - 9.4.1 Mapping Inheritance to the Database 282
 - 9.4.2 STI Considerations 283
 - 9.4.3 STI and Associations 283

- 9.5 Abstract Base Model Classes 286
- 9.6 Polymorphic `has_many` Relationships 287
 - 9.6.1 In the Case of Models with Comments 287
- 9.7 Enums 290
- 9.8 Foreign-Key Constraints 292
- 9.9 Modules for Reusing Common Behavior 292
 - 9.9.1 A Review of Class Scope and Contexts 294
 - 9.9.2 The `included` Callback 295
- 9.10 Modifying Active Record Classes at Runtime 297
 - 9.10.1 Considerations 298
 - 9.10.2 Ruby and Domain-Specific Languages 298
- 9.11 Using Value Objects 299
 - 9.11.1 Immutability 301
- 9.12 Nonpersisted Models 302
- 9.13 PostgreSQL Enhancements 304
 - 9.13.1 Schemaless Data with `hstore` 304
 - 9.13.2 Array Type 306
 - 9.13.3 Network Address Types 308
 - 9.13.4 UUID Type 309
 - 9.13.5 Range Types 310
 - 9.13.6 JSON Type 310
- 9.14 Conclusion 311

Chapter 10 Action View 313

- 10.1 Layouts and Templates 314
 - 10.1.1 Template Filename Conventions 314
 - 10.1.2 Layouts 314
 - 10.1.3 Yielding Content 315
 - 10.1.4 Conditional Output 316
 - 10.1.5 Decent Exposure 317
 - 10.1.6 Standard Instance Variables 318
 - 10.1.7 Displaying `flash` Messages 321
 - 10.1.8 `flash.now` 321
- 10.2 Partial 322
 - 10.2.1 Simple Use Cases 322
 - 10.2.2 Reuse of Partial 324
 - 10.2.3 Shared Partial 324
 - 10.2.4 Passing Variables to Partial 325
 - 10.2.5 Rendering an Object 327

- 10.2.6 Rendering Collections 327
- 10.2.7 Logging 328
- 10.3 Conclusion 329

Chapter 11 All about Helpers 331

- 11.1 ActiveSupportHelper 331
 - 11.1.1 Reporting Validation Errors 332
 - 11.1.2 Automatic Form Creation 335
 - 11.1.3 Customizing the Way Validation Errors Are Highlighted 337
- 11.2 AssetTagHelper 338
 - 11.2.1 Head Helpers 338
 - 11.2.2 Asset Helpers 341
 - 11.2.3 Using Asset Hosts 343
 - 11.2.4 For Plugins Only 345
- 11.3 AtomFeedHelper 346
- 11.4 CacheHelper 348
- 11.5 CaptureHelper 348
- 11.6 CsrHelper 349
- 11.7 DateHelper 349
 - 11.7.1 The Date and Time Selection Helpers 350
 - 11.7.2 The Individual Date and Time Select Helpers 351
 - 11.7.3 Common Options for Date Selection Helpers 354
 - 11.7.4 `distance_in_time` Methods with Complex Descriptive Names 355
 - 11.7.5 `time_tag(date_or_time, *args, &block)` 356
- 11.8 DebugHelper 356
- 11.9 FormHelper 357
 - 11.9.1 Creating Forms for Models 357
 - 11.9.2 How Form Helpers Get Their Values 361
 - 11.9.3 Integrating Additional Objects in One Form 362
 - 11.9.4 Customized Form Builders 366
 - 11.9.5 Form Inputs 366
- 11.10 FormOptionsHelper 371
 - 11.10.1 Select Helpers 371
 - 11.10.2 Check Box/Radio Helpers 373
 - 11.10.3 Option Helpers 375
- 11.11 FormTagHelper 379

11.12	JavaScriptHelper	385
11.13	NumberHelper	385
11.14	OutputSafetyHelper	390
11.15	RecordTagHelper	390
11.16	RenderingHelper	391
11.17	SanitizeHelper	391
11.18	TagHelper	393
11.19	TextHelper	395
11.20	TranslationHelper and the I18n API	399
11.20.1	Localized Views	400
11.20.2	TranslationHelper Methods	400
11.20.3	I18n Setup	401
11.20.4	Setting and Passing the Locale	402
11.20.5	Setting Locale from Client-Supplied Information	405
11.20.6	Internationalizing Your Application	407
11.20.7	Organization of Locale Files	409
11.20.8	Looking Up Translations	410
11.20.9	How to Store Your Custom Translations	413
11.20.10	Overview of Other Built-In Methods That Provide I18n Support	416
11.20.11	Exception Handling	417
11.21	UrlHelper	418
11.22	Writing Your Own View Helpers	422
11.22.1	Small Optimizations: The Title Helper	422
11.22.2	Encapsulating View Logic: The photo_for Helper	423
11.22.3	Smart View: The breadcrumbs Helper	424
11.23	Wrapping and Generalizing Partial	425
11.23.1	A tiles Helper	425
11.23.2	Generalizing Partial	428
11.24	Conclusion	431

Chapter 12 Haml 433

12.1	Getting Started	434
12.2	The Basics	434
12.2.1	Creating an Element	434
12.2.2	Attributes	434
12.2.3	Classes and IDs	436

12.2.4	Implicit Divs	438
12.2.5	Empty Tags	439
12.3	Doctype	440
12.4	Comments	440
12.4.1	HTML Comments	440
12.4.2	Haml Comments	440
12.5	Evaluating Ruby Code	441
12.5.1	Interpolation	442
12.5.2	Escaping/Unescaping HTML	442
12.5.3	Escaping the First Character of a Line	442
12.5.4	Multiline Declarations	443
12.6	Helpers	443
12.6.1	Object Reference []	443
12.6.2	page_class	444
12.6.3	list_of(enum, opts = {}) { item ... }	444
12.7	Filters	444
12.8	Haml and Content	445
12.9	Configuration Options	446
12.9.1	autoclose	446
12.9.2	cdata	446
12.9.3	compiler_class	447
12.9.4	Encoding	447
12.9.5	escape_attrs	447
12.9.6	escape_html	447
12.9.7	format	447
12.9.8	hyphenate_data_attrs	447
12.9.9	mime_type	447
12.9.10	parser_class	447
12.9.11	preserve	448
12.9.12	remove_whitespace	448
12.9.13	ugly	448
12.10	Conclusion	448

Chapter 13 Session Management 449

13.1	What to Store in the Session	450
13.1.1	The Current User	450
13.1.2	Session Use Guidelines	450
13.2	Session Options	451

- 13.3 Storage Mechanisms 451
 - 13.3.1 Active Record Session Store 451
 - 13.3.2 Memcached Session Storage 452
 - 13.3.3 The Controversial CookieStore 453
 - 13.3.4 Cleaning Up Old Sessions 455
- 13.4 Cookies 455
 - 13.4.1 Reading and Writing Cookies 455
- 13.5 Conclusion 457

Chapter 14 Authentication and Authorization 459

- 14.1 Devise 459
 - 14.1.1 Getting Started 460
 - 14.1.2 Modules 460
 - 14.1.3 Models 461
 - 14.1.4 Controllers 462
 - 14.1.5 Views 463
 - 14.1.6 Configuration 463
 - 14.1.7 Strong Parameters 464
 - 14.1.8 Extensions 465
 - 14.1.9 Testing with Devise 465
 - 14.1.10 Summary 466
- 14.2 has_secure_password 466
 - 14.2.1 Getting Started 466
 - 14.2.2 Creating the Models 466
 - 14.2.3 Setting Up the Controllers 467
 - 14.2.4 Controller, Limiting Access to Actions 469
 - 14.2.5 Summary 469
- 14.3 Pundit 470
 - 14.3.1 Getting Started 470
 - 14.3.2 Creating a Policy 471
 - 14.3.3 Controller Integration 472
 - 14.3.4 Policy Scopes 473
 - 14.3.5 Strong Parameters 474
 - 14.3.6 Testing Policies 475
- 14.4 Conclusion 476

Chapter 15 Security 477

- 15.1 Password Management 477
- 15.2 Log Masking 479

- 15.3 SSL (Secure Sockets Layer) 480
- 15.4 Model Mass-Assignment Attributes Protection 481
- 15.5 SQL Injection 483
 - 15.5.1 What Is an SQL Injection? 483
- 15.6 Cross-Site Scripting (XSS) 484
 - 15.6.1 HTML Escaping 485
 - 15.6.2 HTML Sanitization 487
 - 15.6.3 Input versus Output Escaping 487
- 15.7 XSRF (Cross-Site Request Forgery) 487
 - 15.7.1 Restricting HTTP Method for Actions with Side Effects 488
 - 15.7.2 Require Security Token for Protected Requests 489
 - 15.7.3 Client-Side Security Token Handling 490
- 15.8 Session Fixation Attacks 490
- 15.9 Keeping Secrets 491
- 15.10 Conclusion 492

Chapter 16 Action Mailer 493

- 16.1 Setup 493
- 16.2 Mailer Models 494
 - 16.2.1 Preparing Outbound Email Messages 494
 - 16.2.2 HTML Email Messages 496
 - 16.2.3 Multipart Messages 497
 - 16.2.4 Attachments 497
 - 16.2.5 Generating URLs 498
 - 16.2.6 Mailer Layouts 498
 - 16.2.7 Sending an Email 499
 - 16.2.8 Callbacks 499
- 16.3 Receiving Emails 500
 - 16.3.1 Handling Incoming Attachments 501
- 16.4 Server Configuration 502
- 16.5 Testing Email Content 502
- 16.6 Previews 503
- 16.7 Conclusion 504

Chapter 17 Caching and Performance 505

- 17.1 View Caching 505
 - 17.1.1 Page Caching 506
 - 17.1.2 Action Caching 508

- 17.1.3 Fragment Caching 509
- 17.1.4 Russian Doll Caching 514
- 17.1.5 Conditional Caching 516
- 17.1.6 Expiration of Cached Content 516
- 17.1.7 Automatic Cache Expiry with Sweepers 517
- 17.1.8 Avoiding Extra Database Activity 518
- 17.1.9 Cache Logging 519
- 17.1.10 Cache Storage 519
- 17.2 Data Caching 521
 - 17.2.1 Eliminating Extra Database Lookups 521
 - 17.2.2 Initializing New Caches 522
 - 17.2.3 fetch Options 522
- 17.3 Control of Web Caching 523
 - 17.3.1 expires_in(seconds, options =) 523
 - 17.3.2 expires_now 524
- 17.4 ETags 524
 - 17.4.1 fresh_when(options) 525
 - 17.4.2 stale?(options) 525
- 17.5 Conclusion 526

Chapter 18 Background Processing 527

- 18.1 Delayed Job 528
 - 18.1.1 Getting Started 528
 - 18.1.2 Creating Jobs 529
 - 18.1.3 Running 530
 - 18.1.4 Summary 530
- 18.2 Sidekiq 531
 - 18.2.1 Getting Started 531
 - 18.2.2 Workers 532
 - 18.2.3 Scheduled Jobs 533
 - 18.2.4 Delayed Action Mailer 533
 - 18.2.5 Running 534
 - 18.2.6 Error Handling 536
 - 18.2.7 Monitoring 536
 - 18.2.8 Summary 537
- 18.3 Resque 537
 - 18.3.1 Getting Started 537
 - 18.3.2 Creating Jobs 538
 - 18.3.3 Hooks 539

- 18.3.4 Plugins 540
- 18.3.5 Running 540
- 18.3.6 Monitoring 541
- 18.3.7 Summary 541
- 18.4 Rails Runner 541
 - 18.4.1 Getting Started 542
 - 18.4.2 Usage Notes 542
 - 18.4.3 Considerations 543
 - 18.4.4 Summary 543
- 18.5 Conclusion 543

Chapter 19 Ajax on Rails 545

- 19.0.1 Firebug 546
- 19.1 Unobtrusive JavaScript 547
 - 19.1.1 UJS Usage 547
 - 19.1.2 Helpers 548
 - 19.1.3 jQuery UJS Custom Events 550
- 19.2 Turbolinks 551
 - 19.2.1 Turbolinks Usage 551
 - 19.2.2 Turbolinks Events 551
 - 19.2.3 Controversy 552
- 19.3 Ajax and JSON 553
 - 19.3.1 Ajax link_to 553
- 19.4 Ajax and HTML 555
- 19.5 Ajax and JavaScript 557
- 19.6 Conclusion 558

Chapter 20 Asset Pipeline 559

- 20.1 Asset Pipeline 560
- 20.2 Wish List 560
- 20.3 The Big Picture 561
- 20.4 Organization: Where Does Everything Go? 561
- 20.5 Manifest Files 561
 - 20.5.1 Manifest Directives 563
 - 20.5.2 Search Path 564
 - 20.5.3 Gemified Assets 564
 - 20.5.4 Index Files 565
 - 20.5.5 Format Handlers 565
- 20.6 Custom Format Handlers 567

- 20.7 Postprocessing 568
 - 20.7.1 Stylesheets 568
 - 20.7.2 JavaScripts 568
 - 20.7.3 Custom Compressor 569
- 20.8 Helpers 569
 - 20.8.1 Images 570
 - 20.8.2 Getting the URL of an Asset File 570
 - 20.8.3 Built-In SASS Asset Path Helpers 570
 - 20.8.4 Data URIs 571
- 20.9 Fingerprinting 571
- 20.10 Serving the Files 572
 - 20.10.1 GZip Compression 573
- 20.11 Rake Tasks 573
- 20.12 Conclusion 574

Chapter 21 RSpec 575

- 21.1 Introduction 575
- 21.2 Basic Syntax and API 578
 - 21.2.1 describe and context 578
 - 21.2.2 let(:name) { expression } 578
 - 21.2.3 let!(:name) { expression } 580
 - 21.2.4 before and after 580
 - 21.2.5 it 581
 - 21.2.6 specify 581
 - 21.2.7 pending 582
 - 21.2.8 expect(...).to / expect(...).not_to 583
 - 21.2.9 Implicit Subject 586
 - 21.2.10 Explicit Subject 586
- 21.3 Matchers 587
- 21.4 Custom Expectation Matchers 588
 - 21.4.1 Custom Matcher DSL 590
 - 21.4.2 *Fluent* Chaining 590
- 21.5 Shared Behaviors 591
- 21.6 Shared Context 592
- 21.7 RSpec's Mocks and Stubs 592
 - 21.7.1 Test Doubles 592
 - 21.7.2 Null Objects 593
 - 21.7.3 Method Stubs 593
 - 21.7.4 Partial Mocking and Stubbing 594

- 21.7.5 `receive_message_chain` 594
- 21.8 Running Specs 595
- 21.9 RSpec Rails Gem 596
 - 21.9.1 Installation 596
 - 21.9.2 Model Specs 599
 - 21.9.3 Controller Specs 601
 - 21.9.4 View Specs 604
 - 21.9.5 Helper Specs 605
 - 21.9.6 Feature Specs 606
- 21.10 RSpec Tools 609
 - 21.10.1 Guard-RSpec 609
 - 21.10.2 Spring 610
 - 21.10.3 Specjour 610
 - 21.10.4 SimpleCov 610
- 21.11 Conclusion 610

Chapter 22 XML 611

- 22.1 The `to_xml` Method 611
 - 22.1.1 Customizing `to_xml` Output 613
 - 22.1.2 Associations and `to_xml` 614
 - 22.1.3 Advanced `to_xml` Usage 617
 - 22.1.4 Dynamic Runtime Attributes 618
 - 22.1.5 Overriding `to_xml` 620
- 22.2 The XML Builder 620
- 22.3 Parsing XML 622
 - 22.3.1 Turning XML into Hashes 622
 - 22.3.2 Typecasting 623
- 22.4 Conclusion 624

Appendix A Active Model API Reference 625

- A.1 AttributeMethods 625
 - A.1.1 `active_model/attribute_methods.rb` 626
- A.2 Callbacks 627
 - A.2.1 `active_model/callbacks.rb` 628
- A.3 Conversion 629
 - A.3.1 `active_model/conversion.rb` 629
- A.4 Dirty 629
 - A.4.1 `active_model/dirty.rb` 630

A.5	Errors	631
A.5.1	active_model/errors.rb	631
A.6	ForbiddenAttributesError	635
A.7	Lint::Tests	635
A.8	Model	635
A.9	Name	636
A.9.1	active_model/naming.rb	637
A.10	Naming	638
A.10.1	active_model/naming.rb	638
A.11	SecurePassword	638
A.12	Serialization	638
A.12.1	active_model/serialization.rb	639
A.13	Serializers::JSON	639
A.13.1	active_model/serializers/json.rb	639
A.14	Serializers::Xml	639
A.14.1	active_model/serializers/xml.rb	640
A.15	Translation	640
A.15.1	active_model/translation.rb	641
A.16	Validations	641
A.16.1	active_model/validations	642
A.16.2	active_model/validations/absence	643
A.16.3	active_model/validations/acceptance	643
A.16.4	active_model/validations/callbacks	643
A.16.5	active_model/validations/ confirmation	644
A.16.6	active_model/validations/exclusion	644
A.16.7	active_model/validations/format	644
A.16.8	active_model/validations/inclusion	645
A.16.9	active_model/validations/length	645
A.16.10	active_model/validations/ numericality	646
A.16.11	active_model/validations/presence	647
A.16.12	active_model/validations/validates	647
A.16.13	active_model/validations/with	648
A.17	Validator	648
A.17.1	active_model/validator.rb	649

Appendix B Active Support API Reference 651

- B.1 Array 651
 - B.1.1 `active_support/core_ext/array/`
 `access` 651
 - B.1.2 `active_support/core_ext/array/`
 `conversions` 652
 - B.1.3 `active_support/core_ext/array/`
 `extract_options` 655
 - B.1.4 `active_support/core_ext/array/`
 `grouping` 655
 - B.1.5 `active_support/core_ext/array/`
 `prepend_and_append` 656
 - B.1.6 `active_support/core_ext/array/wrap` 657
 - B.1.7 `active_support/core_ext/object/`
 `blank` 657
 - B.1.8 `active_support/core_ext/object/`
 `to_param` 657
- B.2 `ActiveSupport::BacktraceCleaner` 657
 - B.2.1 `active_support/backtrace_cleaner` 657
- B.3 Benchmark 658
 - B.3.1 `ms` 658
- B.4 `ActiveSupport::Benchmarkable` 658
 - B.4.1 `active_support/benchmarkable` 658
- B.5 BigDecimal 659
 - B.5.1 `active_support/core_ext/big_decimal/`
 `conversions` 659
 - B.5.2 `active_support/json/encoding` 659
- B.6 `ActiveSupport::Cache::Store` 660
- B.7 `ActiveSupport::CachingKeyGenerator` 665
 - B.7.1 `active_support/key_generator` 665
- B.8 `ActiveSupport::Callbacks` 665
 - B.8.1 `active_support/callbacks` 666
- B.9 Class 668
 - B.9.1 `active_support/core_ext/class/`
 `attribute` 668

- B.9.2 `active_support/core_ext/class/attribute`
 `_accessors` 670
- B.9.3 `active_support/core_ext/class/attribute`
 `_accessors` 671
- B.9.4 `active_support/core_ext/class/delegating`
 `_attributes` 671
- B.9.5 `active_support/core_ext/class/`
 `subclasses` 671
- B.10 `ActiveSupport::Concern` 671
 - B.10.1 `active_support/concern` 671
- B.11 `ActiveSupport::Concurrency` 672
 - B.11.1 `ActiveSupport::Concurrency::Latch` 672
- B.12 `ActiveSupport::Configurable` 673
 - B.12.1 `active_support/configurable` 673
- B.13 `Date` 673
 - B.13.1 `active_support/core_ext/date/`
 `acts_like` 673
 - B.13.2 `active_support/core_ext/date/`
 `calculations` 674
 - B.13.3 `active_support/core_ext/date/`
 `conversions` 681
 - B.13.4 `active_support/core_ext/date/zones` 682
 - B.13.5 `active_support/json/encoding` 682
- B.14 `DateTime` 682
 - B.14.1 `active_support/core_ext/date_time/`
 `acts_like` 682
 - B.14.2 `active_support/core_ext/date_time/`
 `calculations` 682
 - B.14.3 `active_support/core_ext/date_time/`
 `conversions` 685
 - B.14.4 `active_support/core_ext/date_time/`
 `zones` 686
 - B.14.5 `active_support/json/encoding` 687
- B.15 `ActiveSupport::Dependencies` 687
 - B.15.1 `active_support/dependencies` 687
 - B.15.2 `active_support/dependencies/`
 `autoload` 691

- B.16 ActiveSupport::Deprecation 693
 - B.16.1 active_support/deprecation 693
- B.17 ActiveSupport::DescendantsTracker 694
 - B.17.1 active_support/descendants_tracker 694
- B.18 ActiveSupport::Duration 695
 - B.18.1 active_support/duration 695
- B.19 Enumerable 696
 - B.19.1 active_support/core_ext/enumerable 696
 - B.19.2 active_support/json/encoding 697
- B.20 ERB::Util 697
 - B.20.1 active_support/core_ext/string/output_safety 697
- B.21 FalseClass 698
 - B.21.1 active_support/core_ext/object/blank 698
 - B.21.2 active_support/json/encoding 698
- B.22 File 698
 - B.22.1 active_support/core_ext/file/atomic 698
- B.23 Hash 699
 - B.23.1 active_support/core_ext/hash/compact 699
 - B.23.2 active_support/core_ext/hash/conversions 699
 - B.23.3 active_support/core_ext/hash/deep_merge 700
 - B.23.4 active_support/core_ext/hash/except 700
 - B.23.5 active_support/core_ext/hash/indifferent_access 701
 - B.23.6 active_support/core_ext/hash/keys 701
 - B.23.7 active_support/core_ext/hash/reverse_merge 703
 - B.23.8 active_support/core_ext/hash/slice 703
 - B.23.9 active_support/core_ext/object/to_param 704
 - B.23.10 active_support/core_ext/object/to_query 704
 - B.23.11 active_support/json/encoding 704
 - B.23.12 active_support/core_ext/object/blank 704

- B.24 ActiveSupport::Gzip 704
 - B.24.1 active_support/gzip 705
- B.25 ActiveSupport::HashWithIndifferentAccess 705
 - B.25.1 active_support/
hash_with_indifferent_access 705
- B.26 ActiveSupport::Inflector::Inflections 705
 - B.26.1 active_support/inflector/
inflections 707
 - B.26.2 active_support/inflector/
transliterate 710
- B.27 Integer 711
 - B.27.1 active_support/core_ext/integer/
inflections 711
 - B.27.2 active_support/core_ext/integer/
multiple 712
- B.28 ActiveSupport::JSON 712
 - B.28.1 active_support/json/decoding 712
 - B.28.2 active_support/json/encoding 712
- B.29 Kernel 712
 - B.29.1 active_support/core_ext/kernel/
agnostics 713
 - B.29.2 active_support/core_ext/kernel/
debugger 713
 - B.29.3 active_support/core_ext/kernel/
reporting 713
 - B.29.4 active_support/core_ext/kernel/
singleton_class 714
- B.30 ActiveSupport::KeyGenerator 714
 - B.30.1 active_support/key_generator 714
- B.31 ActiveSupport::Logger 714
 - B.31.1 active_support/logger 714
 - B.31.2 active_support/logger_silence 715
- B.32 ActiveSupport::MessageEncryptor 715
 - B.32.1 active_support/message_encryptor 715
- B.33 ActiveSupport::MessageVerifier 715
 - B.33.1 active_support/message_verifier 716
- B.34 Module 716

- B.34.1 `active_support/core_ext/module/aliasing` 716
- B.34.2 `active_support/core_ext/module/anonymous` 718
- B.34.3 `active_support/core_ext/module/attr_internal` 718
- B.34.4 `active_support/core_ext/module/attribute_accessors` 719
- B.34.5 `active_support/core_ext/module/concerning` 719
- B.34.6 `active_support/core_ext/module/delegation` 720
- B.34.7 `active_support/core_ext/module/deprecation` 722
- B.34.8 `active_support/core_ext/module/introspection` 722
- B.34.9 `active_support/core_ext/module/qualified_const` 723
- B.34.10 `active_support/core_ext/module/reachable` 723
- B.34.11 `active_support/core_ext/module/remove_method` 724
- B.34.12 `active_support/dependencies` 724
- B.35 `ActiveSupport::Multibyte::Chars` 724
 - B.35.1 `active_support/multibyte/chars` 724
 - B.35.2 `active_support/multibyte/unicode` 727
- B.36 `NilClass` 729
 - B.36.1 `active_support/core_ext/object/blank` 729
 - B.36.2 `active_support/json/encoding` 729
- B.37 `ActiveSupport::Notifications` 729
 - B.37.1 `active_support/core_ext/object/blank` 730
 - B.37.2 `active_support/json/encoding` 730
 - B.37.3 `active_support/core_ext/numeric/bytes` 730
 - B.37.4 `active_support/core_ext/numeric/conversions` 731
 - B.37.5 `active_support/core_ext/numeric/time` 737

- B.38 Object 738
 - B.38.1 active_support/core_ext/object/
acts_like 738
 - B.38.2 active_support/core_ext/object/
blank 739
 - B.38.3 active_support/core_ext/object/
deep_dup 739
 - B.38.4 active_support/core_ext/object/
duplicable 739
 - B.38.5 active_support/core_ext/object/
inclusion 740
 - B.38.6 active_support/core_ext/object/instance
_variables 740
 - B.38.7 active_support/core_ext/object/json 741
 - B.38.8 active_support/core_ext/object/
to_param 741
 - B.38.9 active_support/core_ext/object/
to_query 741
 - B.38.10 active_support/core_ext/object/try 741
 - B.38.11 active_support/core_ext/object/
with_options 742
 - B.38.12 active_support/dependencies 742
- B.39 ActiveSupport::OrderedHash 743
 - B.39.1 active_support/ordered_hash 743
- B.40 ActiveSupport::OrderedOptions 743
 - B.40.1 active_support/ordered_options 743
- B.41 ActiveSupport::PerThreadRegistry 744
 - B.41.1 active_support/per_thread_registry 744
- B.42 ActiveSupport::ProxyObject 744
 - B.42.1 active_support/proxy_object 745
- B.43 ActiveSupport::Railtie 745
 - B.43.1 active_support/railtie 745
- B.44 Range 746
 - B.44.1 active_support/core_ext/range/
conversions 746
 - B.44.2 active_support/core_ext/range/each 746
 - B.44.3 active_support/core_ext/range/
include_range 746

- B.44.4 `active_support/core_ext/range/overlaps` 747
- B.44.5 `active_support/core_ext/enumerable` 747
- B.45 `Regexp` 747
 - B.45.1 `active_support/core_ext/regexp` 747
 - B.45.2 `active_support/json/encoding` 747
- B.46 `ActiveSupport::Rescuable` 748
 - B.46.1 `active_support/rescuable` 748
- B.47 `String` 748
 - B.47.1 `active_support/json/encoding` 748
 - B.47.2 `active_support/core_ext/object/blank` 749
 - B.47.3 `active_support/core_ext/string/access` 749
 - B.47.4 `active_support/core_ext/string/behavior` 750
 - B.47.5 `active_support/core_ext/string/conversions` 750
 - B.47.6 `active_support/core_ext/string/exclude` 750
 - B.47.7 `active_support/core_ext/string/filters` 751
 - B.47.8 `active_support/core_ext/string/indent` 752
 - B.47.9 `active_support/core_ext/string/inflections` 752
 - B.47.10 `active_support/core_ext/string/inquiry` 756
 - B.47.11 `active_support/core_ext/string/multibyte` 756
 - B.47.12 `active_support/core_ext/string/output_safety` 757
 - B.47.13 `active_support/core_ext/string/starts_ends_with` 757
 - B.47.14 `active_support/core_ext/string/strip` 757
 - B.47.15 `active_support/core_ext/string/in_time_zone` 758

- B.48 ActiveSupport::StringInquirer 758
- B.49 Struct 758
 - B.49.1 active_support/core_ext/struct 758
- B.50 ActiveSupport::Subscriber 758
- B.51 Symbol 759
 - B.51.1 active_support/json/encoding 759
- B.52 ActiveSupport::TaggedLogging 759
 - B.52.1 active_support/tagged_logger 759
- B.53 ActiveSupport::TestCase 759
 - B.53.1 active_support/test_case 759
- B.54 ActiveSupport::Testing::Assertions 761
 - B.54.1 active_support/testing/assertions 761
 - B.54.2 active_support/testing/time_helpers 762
- B.55 Thread 762
 - B.55.1 active_support/core_ext/thread 763
- B.56 Time 763
 - B.56.1 active_support/json/encoding 763
 - B.56.2 active_support/core_ext/time/
acts_like 764
 - B.56.3 active_support/core_ext/time/
calculations 764
 - B.56.4 active_support/core_ext/time/
conversions 770
 - B.56.5 active_support/core_ext/time/
marshal 771
 - B.56.6 active_support/core_ext/time/zones 771
- B.57 ActiveSupport::TimeWithZone 773
- B.58 ActiveSupport::TimeZone 774
 - B.58.1 active_support/values/time_zone 775
- B.59 TrueClass 778
 - B.59.1 active_support/core_ext/object/
blank 778
 - B.59.2 active_support/json/encoding 778
- B.60 ActiveSupport::XmlMini 778
 - B.60.1 active_support/xml_mini 778

Appendix C Rails Essentials 781

- C.1 Environmental Concerns 781
 - C.1.1 Operating System 781
 - C.1.2 Aliases 782
- C.2 Essential Gems 782
 - C.2.1 Better Errors 782
 - C.2.2 Country Select 783
 - C.2.3 Debugger 783
 - C.2.4 Draper 783
 - C.2.5 Kaminari 784
 - C.2.6 Nested Form Fields 785
 - C.2.7 Pry and Friends 786
 - C.2.8 Rails Admin 787
 - C.2.9 Simple Form 788
 - C.2.10 State Machine 788
- C.3 Ruby Toolbox 789
- C.4 Screencasts 789
 - C.4.1 Railcasts 790

Index 791

This page intentionally left blank

Foreword

A long time ago, I was an intern at a technology company. We had “deploy week,” meaning that after deploying, we took an entire week to fight fires. Moving our code to the production environment would inevitably cause unexpected changes. One day, I read a blog post titled “Unit Testing with Ruby on Rails,” and my life was forever changed. I excitedly went and told my team that we could write code to check whether our code worked before deploying, but they weren’t particularly interested. A few months later, when a friend asked me to be the CTO of his startup, I said, “Only if I can do it in Ruby on Rails.”

My story was fairly typical for that period. I didn’t know anything about Ruby, but I *had* to write my application in Rails. I figured out enough Ruby to fake it and cobbled together an application in record time. There was just one problem: I didn’t really understand how it actually *worked*. This is the deal everyone makes with Rails at the start. You can’t think about the details too much because you’re flying to the sky like a rocket.

This book, however, isn’t about that. When I read *The Rails Way* for the first time, I felt like I truly understood Rails for the first time. All those details I didn’t fully understand were now able to be grokked. Every time someone said, “Rails is magic,” I would smile to myself. If Rails was magic, I had peered behind the curtain. One day, I decided that I should write some documentation to help dispel those kinds of comments. One commit became two; two became twenty. Eventually, I was a large contributor in my own right. Such a long way for someone who had just a few short years earlier never heard of a unit test!

As Rails has changed, so has *The Rails Way*. In fact, one criticism you could make of this book is that it’s not actually “the Rails way”; after all, it teaches you HAML

instead of ERB! I think that this criticism misses the mark. After all, it's not 2005 anymore. To see what I mean, go read the two forewords from the previous edition. They appear right after this one ... I'll wait.

Done? David's foreword was quite accurate for both Rails 2 and *The Rails Way*. At that time, Rails was very much "not as a blank slate equally tolerant of every kind of expression." Rails was built for what I call the "Omakase Stack": you have no choice, you get exactly what Chef David wants to serve you.¹

Yehuda's foreword was also quite accurate—but for Rails 3 and *The Rails™ 3 Way*. "We brought this philosophy to every area of Rails 3: flexibility without compromise." With Rails 3, you get the Omakase stack by default, but you are free to swap out components: if you don't like sushi, you can substitute some sashimi.

There was a lot of wailing and gnashing of teeth during the development of Rails 3. Jeremy Ashkenas called it "by far the worst misfortune to ever happen to Rails." Rails 3 was an investment in the future of Rails, and investments can take a while to pay off. At the release of Rails 3, it seemed like we had waited more than a year for no new features. Rails was a little better but mostly the same. The real benefit was where it couldn't be seen: in the refactoring work. Rails 1 was "red-green." Rails 2 was "red-green." Rails 3 was "refactor." It took a little while for gem authors to take advantage of this flexibility, but eventually, they did.

And that brings us to Rails 4 and *The Rails™ 4 Way*. This book still explains quite a bit about how Rails works at a low level but also gives you an alternate vision from the Omakase Stack, based on the experience and talent of Hashrocket. In many ways, *The Rails™ 4 Way*, *Agile Web Development with Rails*, and *Rails 4 in Action* are all "the Rails way." Contemporary Rails developers get the best of both worlds: They can take advantage of the rapid development of convention over configuration, but if they choose to follow a different convention, they can. And we have many sets of conventions to choose from. It's no longer "David's way or the highway," though David's way is obviously the default, as it should be.

It has been an amazing few years for Rails, and it has been a pleasure to take a part in its development. I hope that this book will give you the same level of insight and clarity into Rails as it did for me, years ago, while also sparking your imagination for what Rails will undoubtedly become in the future.

—Steve Klabnik

1. *Omakase* is a Japanese term used at sushi restaurants to leave the selection to the chef. To learn more about the Omakase stack, read <http://words.steveklabnik.com/rails-has-two-default-stacks>

Foreword to the Previous Edition

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

To be able to accept that trade, you need to understand not just how to do something in Rails but also why it's done like that. Only by understanding the why will you be able to consistently work with the framework instead of against it. It doesn't mean that you'll always have to agree with a certain choice, but you will need to agree to the overarching 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, but 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 and 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 to the Previous Edition

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 development-friendly 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 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 nonrelational 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 starting 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™ 3 Way* is a comprehensive resource that digs into the new features in Rails 3 and perhaps, more important, the rationale behind them.

—Yehuda Katz
Rails Core

Introduction

It's an exciting time for the Rails community. We have matured tremendously and our mainstream adoption continues to pick up steam. Nearly 10 years after DHH first started playing with Ruby, it's safe to say that Rails remains a relevant and vital tool in the greater web technology ecosystem.

Rails 4 represents a big step forward for the community. We shed a variety of vestigial features that had been deprecated in Rails 3. Security was beefed up and raw performance improved. Most everything in the framework feels, well, tighter than before. Rails 4 is leaner and meaner than its previous incarnations, and so is this edition of *The Rails Way*.

In addition to normal revisions to bring the text up to date with the evolution of Rails' numerous APIs, this edition adds a significant amount of new and updated material about security, performance, and caching; Haml; RSpec; Ajax; and the new Asset Pipeline.

About This Book

As with previous editions, this book is not a tutorial or basic introduction to Ruby or Rails. It is meant as a day-to-day reference for the full-time Rails developer. 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 and 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 API documentation, in-depth explanations, and relevant examples.

Book Structure

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

Believe me, it has not been easy coming up with a structure that makes perfect sense for everyone. Particularly, I have noted surprise in some readers when they notice that 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.

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 justifying 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 (...) often 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 parts of it verbatim in your code, I supply a listing heading. There are not too many of those. The whole set of code listings will not add up to a complete working system, nor are there 30 pages of sample application code in an appendix.

The code listings should serve as inspiration for your production-ready work, but keep in mind that it often lacks touches necessary in real-world work. For example, examples of controller code are often missing pagination and access control logic, because it would detract from the point being expressed.

Some of the source code for my examples can be found at https://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 through it.

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 to call 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, I don't cover the built-in Rails functionality (pagination is a good example).

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

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

Recommended Reading and Resources

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

Most Ruby programmers always have their copy of the "Pickaxe" book nearby, *Programming Ruby* (ISBN: 0-9745140-5-5), because it is a good language reference.

Readers interested in really understanding all the nuances of Ruby programming should acquire *The Ruby Way, Second Edition* (ISBN: 0-672-3288-4-4).

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>.

Regarding David Heinemeier Hansson, a.k.a. DHH: I had the pleasure of establishing a friendship with David, 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 my 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.

I don't know if this is true anymore, but back when I wrote the original edition of this book, David had 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 the following:

- *Yehuda* Katz
- *Jamis* Buck
- *Xavier* Noria
- *Tim* Pope

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 to the end unless they're expanding their basic knowledge of the Rails framework. Whatever the case may be, over time, I hope this book gives you as an application developer/programmer greater confidence in making design and implementation decisions while working on your day-to-day tasks. After spending time with this book, your understanding of the fundamental concepts of Rails coupled with hands-on experience should leave you feeling comfortable working on real-world Rails projects, with real-world demands.

If you are in an architectural or development lead role, this book is not targeted to you but should make you feel more comfortable discussing the pros and cons of

Ruby on Rails adoption and ways to extend Rails to meet the particular needs of the project under your direction.

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

Prerequisites

The reader is assumed to have the following knowledge:

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

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

Required Technology

A late-model Apple MacBook Pro running Mac OS X should be fine—just kidding, of course. Linux is pretty good for Rails development also. Microsoft Windows—well, let me just put it this way: your mileage may vary. I’m being nice and diplomatic in saying that. We specifically do not discuss Rails development on Microsoft platforms in this book. It’s common knowledge that the vast majority of working Rails professionals develop and deploy on non-Microsoft platforms.

This page intentionally left blank

Acknowledgments

The Rails™ 4 Way was very much a team effort. On behalf of myself and Kevin, I would like to thank Vitaly Kushner and Ari Lerner for their contributions and support throughout the life of the project. We'd also like to thank Mike Perham, Juanito Fatas, Phillip Campbell, Brian Cardarella, Carlos Souza, and Michael Mazyar for technical review and edits. I must thank my understanding business partner Trevor Owens and staff at Lean Startup Machine for their ongoing support. Of course, Kevin and I also thank our families for their patience as writing tasks quite often ate into our personal time with them.

As always, I'd also like to express a huge debt of gratitude to our executive editor at Pearson: Debra Williams Cauley. Without her constant support and encouragement throughout the years, the Professional Ruby Series would not exist.

—Obie Fernandez
December 2013

This page intentionally left blank

About the Authors

Obie Fernandez

Obie has been hacking computers since he got his first Commodore VIC-20 in the eighties. In the midnineties, he found himself in the right place and time as a programmer on some of the first Java enterprise projects. He moved to Atlanta, Georgia, in 1998 and founded the Extreme Programming (later Agile Atlanta) User Group and was that group's president and organizer for several years. In 2004, he joined world-renowned consultancy ThoughtWorks and made a name for himself tackling high-risk, progressive projects in the enterprise, including some of the first enterprise projects in the world utilizing Ruby on Rails.

As founder and CEO of Hashrocket, one of the world's best web design and development consultancies, Obie specialized in orchestrating the creation of large-scale, web-based applications, both for startups and mission-critical enterprise projects. In 2010, Obie sold his stake in Hashrocket and has been working with technology startups ever since. He's currently cofounder and CTO of Lean Startup Machine, where he leads an awesome technology team and is building recognition as a thought leader on lean startup topics.

Obie's evangelization of Ruby on Rails online via blog posts and publications dates back to early 2005, and it earned him quite a bit of notoriety (and trash talking) from his old friends in the Java open-source community. Since then, he has traveled around the world relentlessly promoting Rails at large industry conferences. The previous two editions of this book are considered the "bibles" of Ruby on Rails development and are bestsellers.

Obie still gets his hands dirty with code daily and posts regularly on various topics to his popular weblog at <http://blog.obiefernandez.com>. His next book, *The Lean Enterprise*, is scheduled to be published in spring 2014.

Kevin Faustino

Kevin is founder and chief craftsman of Remarkable Labs, based in Toronto, Canada. He believes that software should not just work but be well crafted. He founded Remarkable Labs because he wanted to build a company that he would be proud to work for and that other companies would love to work with.

Following his passion for sharing knowledge, Kevin also founded the Toronto Ruby Brigade, which hosts tech talks, hack nights, and book clubs. Kevin has been specializing in Ruby since 2008 and has been professionally developing since 2005.

CHAPTER 9

Advanced Active Record

Respectful debate, honesty, passion, and working systems created an environment that not even the most die-hard enterprise architect could ignore, no matter how buried in Java design patterns. Those who placed technical excellence and pragmatism above religious attachment and vendor cronyism were easily convinced of the benefits that broadening their definition of acceptable technologies could bring.

—Ryan Tomayko¹

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, 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 to define and chain query criteria in a declarative and reusable manner.

1. <http://lesscode.org/2006/03/12/someone-tell-gosling/>

```
1 class Timesheet < ActiveRecord::Base
2   scope :submitted, -> { where(submitted: true) }
3   scope :underutilized, -> { where('total_hours < 40') }
```

To declare a scope, use the `scope` class method, passing it a name as a symbol and a callable object that includes a query criteria within. You can simply use Arel criteria methods such as `where`, `order`, and `limit` to construct the definition as shown in the example. The queries defined in a scope are only evaluated whenever the scope is invoked.

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

Invoke scopes as you would class methods.

```
>> User.delinquent
=> [#<User id: 2, timesheets_updated_at: "2013-04-20 20:02:13"...>]
```

Note that instead of using the scope macro-style method, you can simply define a class method on an Active Record model that returns a scoped method, such as `where`. To illustrate, the following class method is equivalent to the `delinquent` scope defined in the previous example.

```
1 def self.delinquent
2   where('timesheets_updated_at < ?', 1.week.ago)
3 end
```

9.1.1 Scope Parameters

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

```
1 class BillableWeek < ActiveRecord::Base
2   scope :newer_than, ->(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 benefits of scopes is that you can chain them together to create complex queries from simple ones:

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

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 the `underutilized` scope to only submitted timesheets:

```
1 class Timesheet < ActiveRecord::Base
2   scope :submitted, -> { where(submitted: true) }
3   scope :underutilized, -> { submitted.where('total_hours < 40') }
```

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, username: "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.

```
1 scope :tardy, -> {
2   joins(:timesheets).
3   where("timesheets.submitted_at <= ?", 7.days.ago).
4   group("users.id")
5 }
```

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 <= '2013-04-13 18:16:15.203293')
 GROUP BY users.id" # query formatted nicely for the book
```

Note that as demonstrated in the example, it's a good idea to use unambiguous column references (including the 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 to fix it. First, we put the late logic where it belongs—in `Timesheet`:

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

Then we use our new late scope in `tardy`:

```
scope :tardy, -> {
  joins(:timesheets).group("users.id").merge(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 that 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.pluck(: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 `all` method.

```
>> timesheets = Timesheet.all.order("submitted_at DESC")
=> #<ActiveRecord::Relation [#<Timesheet id: 1, status: "open">]>
>> timesheets.where(name: "Durrant Jordan").to_a
=> []
```

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
>>   Timesheet.first
>> 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").to_a
=> [#<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.pluck(:total_hours)
=> [37, 38]

>> u.timesheets.underutilized.update_all("total_hours = total_hours + 2")
=> 2

>> u.timesheets.underutilized.pluck(:total_hours)
=> [39]
```

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 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 Arel underpinnings of Active Record are tremendously powerful and truly elevate the Rails 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 the 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 those scenarios in this section, but first let's get a taste of what a callback looks like. Check out the following silly example:

```
1 class Beethoven < ActiveRecord::Base  
2   before_destroy :last_words  
3  
4   protected  
5  
6   def last_words  
7     logger.info "Friends applaud, the comedy is over"  
8   end  
9 end
```

So prior to dying (ehrm, being destroyed), 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 One-Liners

Now if (and only if) your callback routine is really short,² 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
```

Since 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.

```
1 class Account < ActiveRecord::Base
2   before_destroy { self.update_attribute(:deleted_at, Time.now); false }
3   ...
end
```

9.2.2 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.

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

9.2.3 Matched **before/after** Callbacks

In total, there are 19 types of callbacks you can register on your models! Thirteen of them are matching before/after callback pairs, such as `before_validation` and `after_validation`. Four of them are around callbacks, such as `around_save`. (The other two, `after_initialize` and `after_find`, are special, and we'll discuss them later in this section.)

9.2.3.1 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.)

2. 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.

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_create` (for new records) and `before_update` (for existing records)
- `around_create` (for new records) and `around_update` (for existing records)
- `after_create` (for new records) and `after_update` (for existing records)
- `after_save`

Delete operations have their own callbacks:

- `before_destroy`
- `around_destroy`, which executes a `DELETE` database statement on `yield`
- `after_destroy`, which is called after record has been removed from the database and all attributes have been frozen (`readonly`)

Callbacks may be limited to specific Active Record life cycles (`:create`, `:update`, `:destroy`) by explicitly defining which ones can trigger it using the `:on` option. The `:on` option may accept a single lifecycle (like `on: :create`) or an array of life cycles (like `on: [:create, :update]`).

```
# Run only on create.
before_validation :some_callback, on: :create
```

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 due to the nature of transactions—it’s a bad idea to be able to interfere with the actual operation itself.

- `after_commit`
- `after_rollback`
- `after_touch`

Skipping Callback Execution

The following Active Record methods, when executed, do not run any callbacks:

- decrement
 - decrement_counter
 - delete
 - delete_all
 - increment
 - increment_counter
 - toggle
 - touch
 - update_column
 - update_columns
 - update_all
 - update_counters
-

9.2.4 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 since 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.

9.2.5 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.

9.2.5.1 Cleaning Up Attribute Formatting with **before_validation** 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:

```
1 class CreditCard < ActiveRecord::Base
2   before_validation on: :create do
3     # Strip everything in the number except digits.
4     self.number = number.gsub(/[^0-9]/, "")
5   end
6 end
```

9.2.5.2 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.³

As is often the case, the wording of the requirement itself points you in the direction of the `before_save` callback:

```
1 class Address < ActiveRecord::Base
2
3   before_save :geocode
4   validates_presence_of :street, :city, :state, :country
5   ...
6
7   def to_s
8     [street, city, state, country].compact.join(', ')
9   end
10
11   protected
12
13   def geocode
14     result = Geocoder.coordinates(to_s)
15     self.latitude = result.first
16     self.longitude = result.last
17   end
18 end
```

Note

For the sake of this example, we will not be using Geocoder's Active Record extensions.

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:

```
1 def geolocate
2   result = Geocoder.coordinates(to_s)
3   return false if result.empty? # halt execution
4
5   self.latitude = result.first
6   self.longitude = result.last
7 end
```

3. I recommend the excellent Geocoder gem available at <http://www.rubygeocoder.com>

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:

```
1 def geolocate
2   result = Geocoder.coordinates(to_s)
3   if result.present?
4     self.latitude = result.first
5     self.longitude = result.last
6   else
7     errors[:base] << "Geocoding failed. Please check address."
8     false
9   end
10 end
```

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

9.2.5.3 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` datetime column.

```
1 class Account < ActiveRecord::Base
2   before_destroy do
3     self.update_attribute(:deleted_at, Time.current)
4     false
5   end
6
7   ...
8 end
```

After the `deleted_at` column is populated with the current time, we return `false` in the callback to halt execution. This ensures that the underlying record is not actually deleted from the database.⁴

It's probably worth mentioning that there are ways that Rails allows you to unintentionally circumvent `before_destroy` callbacks:

4. Real-life implementation of the example would also need to modify all finders to include conditions where `deleted_at` is null; 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 `destroyed_at` created by Dockyard that does exactly that, and you can find it at https://github.com/dockyard/destroyed_at

- 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.
- 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.

9.2.5.4 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 Thoughtbot's Paperclip⁵ gem is a good example:

```
1 # Destroys the file. Called in an after_destroy callback.
2 def destroy_attached_files
3   Paperclip.log("Deleting attachments.")
4   each_attachment do |name, attachment|
5     attachment.send(:flush_deletes)
6   end
7 end
```

9.2.6 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 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:

```
1 after_initialize do
2   if new_record?
```

5. Get Paperclip at <https://github.com/thoughtbot/paperclip>

```
3     ...
4   end
5 end
```

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:

```
1 class User < ActiveRecord::Base
2   serialize :preferences # defaults to nil
3   ...
4
5   protected
6
7   def after_initialize
8     self.preferences ||= Hash.new
9   end
10 end
```

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`.

Kevin Says ...

You could change the previous example to not use callbacks by using the Active Record `store`, a wrapper around `serialize` that is used exclusively for storing hashes in a database column.

```
1 class User < ActiveRecord::Base
2   serialize :preferences # defaults to nil
3   store :preferences, accessors: [:show_help_text]
4   ...
5 end
```

By default, the `preferences` attribute would be populated with an empty hash. Another added benefit is the ability to explicitly define accessors, removing the need to interact with the underlying hash directly. To illustrate, let's set the `show_help_text` preference to true:

```
>> user = User.new
=> #<User id: nil, properties: {}, ...>
```

```
>> user.show_help_text = true
=> true
>> user.properties
=> {"show_help_text" => true}
```

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."

9.2.7 Callback Classes

It is common enough to want to reuse callback code for more than one object that Rails provides 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:

```
1 class MarkDeleted
2   def self.before_destroy(model)
3     model.update_attribute(:deleted_at, Time.current)
4     false
5   end
6 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:

```
1 class Account < ActiveRecord::Base
2   before_destroy MarkDeleted
3   ...
4 end
5
6 class Invoice < ActiveRecord::Base
7   before_destroy MarkDeleted
8   ...
9 end
```

9.2.7.1 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:

```
1 class Auditor
2   def initialize(audit_log)
3     @audit_log = audit_log
4   end
5
6   def after_create(model)
7     @audit_log.created(model.inspect)
8   end
9
10  def after_update(model)
11    @audit_log.updated(model.inspect)
12  end
13
14  def after_destroy(model)
15    @audit_log.destroyed(model.inspect)
16  end
17 end
```

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

```
1 class Account < ActiveRecord::Base
2   after_create Auditor.new(DEFAULT_AUDIT_LOG)
3   after_update Auditor.new(DEFAULT_AUDIT_LOG)
4   after_destroy Auditor.new(DEFAULT_AUDIT_LOG)
5   ...
6 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, allowing you to 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.

Listing 9.1 A Quick-and-Dirty `acts_as_audited` Method

```
1 class ActiveRecord::Base
2   def self.acts_as_audited(audit_log=DEFAULT_AUDIT_LOG)
3     auditor = Auditor.new(audit_log)
4     after_create auditor
5     after_update auditor
6     after_destroy auditor
7   end
8 end
```

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

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

9.2.7.2 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 easy to test in isolation.

```
1 describe '#after_create' do
2   let(:auditable) { double() }
3   let(:log) { double() }
4   let(:content) { 'foo' }
5
6   it 'audits a model was created' do
7     expect(auditable).to receive(:inspect).and_return(content)
8     expect(log).to receive(:created).and_return(content)
9     Auditor.new(log).after_create(auditable)
10  end
11 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.

Calculation methods can be used in combination with Active Record relation methods to customize the query. Since calculation methods do not return an `ActiveRecord::Relation`, they must be the last method in a scope chain.

There are two basic forms of output:

Single Aggregate Value The single value is typecast 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 examples illustrate the usage of various calculation methods.

```
1 Person.calculate(:count, :all) # the same as Person.count
2
3 # SELECT AVG(age) FROM people
4 Person.average(:age)
5
```

```

6 # Selects the minimum age for everyone with a last name other than "Drake."
7 Person.where.not(last_name: 'Drake').minimum(:age)
8
9 # Selects the minimum age for any family without any minors.
10 Person.having('min(age) > 17').group(:last_name).minimum(:age)

```

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 column present.

9.3.3 **ids**

Return all the `ids` for a relation based on its table's primary key.

```
User.ids # SELECT id FROM "users"
```

9.3.4 **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.

9.3.5 **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.

9.3.6 **pluck(*column_names)**

The `pluck` method queries the database for one or more columns of the underlying table of a model.

```

>> User.pluck(:id, :name)
=> [[1, 'Obie']]
>> User.pluck(:name)
=> ['Obie']

```

It returns an array of values of the specified columns with the corresponding data type.

9.3.7 **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 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 the 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.

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:

```
1 class Timesheet < ActiveRecord::Base
2   ...
3
4   def billable_hours_outstanding
5     if submitted?
6       billable_weeks.map(&:total_hours).sum
7     else
8       0
9     end
10  end
11
12  def self.billable_hours_outstanding_for(user)
13    user.timesheets.map(&:billable_hours_outstanding).sum
14  end
15
16 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:

```
1 def billable_hours_outstanding
2   if submitted? && not paid?
3     billable_weeks.map(&:total_hours).sum
4   else
5     0
6   end
7 end
```

That latest change is a clear violation of the open-closed principle,⁶ 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`.

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.

```
1 class Timesheet < ActiveRecord::Base
2   # nonrelevant code omitted
3
4   def self.billable_hours_outstanding_for(user)
5     user.timesheets.map(&:billable_hours_outstanding).sum
6   end
7 end
8
9 class DraftTimesheet < Timesheet
10  def billable_hours_outstanding
11    0
12  end
13 end
14
15 class SubmittedTimesheet < Timesheet
16  def billable_hours_outstanding
17    billable_weeks.map(&:total_hours).sum
18  end
19 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!

```
1 class PaidTimesheet < Timesheet
2   def billable_hours_outstanding
3     billable_weeks.map(&:total_hours).sum - paid_hours
4   end
5 end
```

6. http://en.wikipedia.org/wiki/Open/closed_principle has a good summary.

9.4.1 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 hold all 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:

```
1 class AddTypeToTimesheet < ActiveRecord::Migration
2   def change
3     add_column :timesheets, :type, :string
4   end
5 end
```

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 query 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.first
=> #<DraftTimesheet:0x2212354...>
```

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.)

9.4.2 STI Considerations

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

First, 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 that should be a problem unless you've made some pretty bad data-modeling decisions.

Second and more important, 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 setter method `inheritance_column` in the base class. For the `Timesheet` example, we could do the following:

```
1 class Timesheet < ActiveRecord::Base
2   self.inheritance_column = 'object_type'
3 end
```

Now Rails will automatically populate the `object_type` column with the object's type.

9.4.3 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:⁷

Listing 9.2 The Places Database Schema and the Place Class

```

1 # == Schema Information
2 #
3 # Table name: places
4 #
5 #   id :integer(11) not null, primary key
6 #   region_id :integer(11)
7 #   type :string(255)
8 #   name :string(255)
9 #   description :string(255)
10 #   latitude :decimal(20, 1)
11 #   longitude :decimal(20, 1)
12 #   population :integer(11)
13 #   created_at :datetime
14 #   updated_at :datetime
15
16 class Place < ActiveRecord::Base
17 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 and define concrete subclasses of `Place`:

```

1 class State < Place
2   has_many :counties, foreign_key: 'region_id'
3 end
4
5 class County < Place
6   belongs_to :state, foreign_key: 'region_id'
7   has_many :cities, foreign_key: 'region_id'
8 end
9
10 class City < Place
11   belongs_to :county, foreign_key: 'region_id'
12 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:

7. For autogenerated schema information added to the top of your model classes, try the `annotate` gem at https://github.com/ctran/annotate_models

```

1 class State < Place
2   has_many :counties, foreign_key: 'region_id'
3   has_many :cities, through: :counties
4 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`.

```

1 class State < Place
2   has_many :counties
3   has_many :cities
4 end
5
6 class County < Place
7   belongs_to :state

```

```
8   has_many :cities
9 end
10
11 class City < Place
12   belongs_to :county
13 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, that will make the purists positively hate you.

9.5 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`.⁸

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 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

```
1 class Place < ActiveRecord::Base
2   self.abstract_class = true
3 end
```

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.

8. <http://m.onkey.org/namespaced-models>

9.6 Polymorphic has_many Relationships

Rails gives you the ability to make one class `belongs_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.⁹

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

9.6.1 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.

```
1 class Comment < ActiveRecord::Base
2   belongs_to :timesheet
3   belongs_to :expense_report
4 end
```

I call that approach 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

9. <http://techspot.zzzeek.org/2007/05/29/polymorphic-associations-with-sqlalchemy/>

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.”

9.6.1.1 The Interface

Using a polymorphic association, we need to 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.

```
1 class Comment < ActiveRecord::Base
2   belongs_to :commentable, polymorphic: true
3 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:

```
1 class Timesheet < ActiveRecord::Base
2   has_many :comments, as: :commentable
3 end
4
5 class BillableWeek < ActiveRecord::Base
6   has_many :comments, as: :commentable
7 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.

9.6.1.2 The Database Columns

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

```
1 class CreateComments < ActiveRecord::Migration
2   def change
3     create_table :comments do |t|
4       t.text :body
5       t.integer :commentable
```

```

6      t.string :commentable_type
7    end
8  end
9 end

```

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:

```

1 create_table :comments do |t|
2   t.text :body
3   t.references :commentable, polymorphic: true
4 end

```

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

```

>> c = Comment.create(body: '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.

9.6.1.3 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:

```

1 class Comment < ActiveRecord::Base
2   belongs_to :user # author of the comment
3   belongs_to :commentable, polymorphic: true
4 end
5
6 class User < ActiveRecord::Base

```

```

7   has_many :comments
8   has_many :commentables, through: :comments
9 end
10
11 >> User.first.commentables
12 ActiveRecord::HasManyThroughAssociationPolymorphicSourceError: Cannot
13 have a has_many :through association 'User#commentables' on the polymorphic object

```

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:

```

1 class User < ActiveRecord::Base
2   has_many :comments
3   has_many :commented_timesheets, through: :comments,
4     source: :commentable, source_type: 'Timesheet'
5   has_many :commented_billable_weeks, through: :comments,
6     source: :commentable, source_type: 'BillableWeek'
7 end

```

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

```

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

```

9.7 Enums

One of the newest additions to Active Record introduced in Rails 4.1 is the ability to set an attribute as an enumerable. Once an attribute has been set as an enumerable, Active Record will restrict the assignment of the attribute to a collection of predefined values.

To declare an enumerable attribute, use the `enum` macro-style class method, passing it an attribute name and an array of status values that the attribute can be set to.

```

1 class Post < ActiveRecord::Base
2   enum status: %i(draft published archived)
3   ...
4 end

```

Active Record implicitly maps each predefined value of an enum attribute to an integer; therefore, the column type of the enum attribute must be an integer as well. By default, an enum attribute will be set to `nil`. To set an initial state, one can set a default value in a migration. It's recommended to set this value to the first declared status, which would map to 0.

```

1 class CreatePosts < ActiveRecord::Migration
2   def change
3     create_table :posts do |t|
4       t.integer :status, default: 0
5     end
6   end
7 end

```

For instance, given our example, the default status of a `Post` model would be “draft”:

```

>> Post.new.status
=> "draft"

```

You should never have to work with the underlying integer data type of an enum attribute, as Active Record creates both predicate and bang methods for each status value.

```

1 post.draft!
2 post.draft?      # => true
3 post.published?  # => false
4 post.status      # => "draft"
5
6 post.published!
7 post.published?  # => true
8 post.draft?      # => false
9 post.status      # => "published"
10
11 post.status = nil
12 post.status.nil? # => true
13 post.status      # => nil

```

Active Record also provides scope methods for each status value. Invoking one of these scopes will return all records with that given status.

```

Post.draft
# Post Load (0.1ms)  SELECT "posts".* FROM "posts"
  WHERE   "posts"."status" = 0

```

Note

Active Record creates a class method with a pluralized name of the defined enum on the model that returns a hash with the key and value of each status. In our preceding example, the `Post` model would have a class method named `statuses`.

```

>> Post.statuses
=> {"draft"=>0, "published"=>1, "archived"=>2}

```

You should only need to access this class method when you need to know the underlying ordinal value of an enum.

With the addition of the `enum` attribute, Active Record finally has a simple state machine out of the box. This feature alone should simplify models that had previously depended on multiple boolean fields to manage state. If you require more advanced functionality, such as status transition callbacks and conditional transitions, it's still recommended to use a full-blown state machine like `state_machine`.¹⁰

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 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" in this chapter, 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

10. https://github.com/pluginaweek/state_machine

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 and then by the administrator.

Again, here is the polymorphic `has_many :comments, as: :commentable` that we used as the foundation for this functionality:

```
1 class Timesheet < ActiveRecord::Base
2   has_many :comments, as: :commentable
3 end
4
5 class ExpenseReport < ActiveRecord::Base
6   has_many :comments, as: :commentable
7 end
8
9 class Comment < ActiveRecord::Base
10  belongs_to :commentable, polymorphic: true
11 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.

```
1 class Comment < ActiveRecord::Base
2   scope :recent, -> { order('created_at desc').limit(10) }
3 end
4
5 class CommentsController < ApplicationController
6   before_action :require_admin, only: :recent
7   expose(:recent_comments) { Comment.recent }
8 end
```

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

```
1 %ul.recent.comments
2   - recent_comments.each do |comment|
3     %li.comment
4       %h4= comment.created_at
5       = comment.text
6       .meta
7       Comment on:
8       = link_to comment.commentable.title, comment.commentable
9       # 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 the unreviewed comments for an item,

we can use a named scope on the `Comment` class together with the comments association.

```
1 class Comment < ActiveRecord::Base
2   scope :unreviewed, -> { where(reviewed: false) }
3 end
4
5 >> 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:

```
1 module Commentable
2   has_many :comments, as: :commentable
3 end
4
5 class Timesheet < ActiveRecord::Base
6   include Commentable
7 end
8
9 class ExpenseReport < ActiveRecord::Base
10  include Commentable
11 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.

9.9.1 A Review of Class Scope and Contexts

In many other interpreted, object-oriented 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," and the second 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, we 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 Foo` at line 4, we are once again telling the interpreter to execute the following code in the context of the `Foo` 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, let's say we defined our `Commentable` module like this:

```
1 module Commentable
2   has_many :comments, as: :commentable
3 end
```

In this case, 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.

9.9.2 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`, etc.):

```
1 module Commentable
2   def self.included(base)
3     base.class_eval do
4       has_many :comments, as: :commentable
5     end
6   end
7 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 gems, that it was added as a first-class concept in the ActiveSupport API as of Rails 3. The previous example becomes shorter and easier to read as a result:

```
1 # app/models/concerns/commentable.rb
2 module Commentable
3   extend ActiveSupport::Concern
4   included do
5     has_many :comments, as: :commentable
6   end
7 end
```

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

As of version 4.0, Rails includes the directory `app/models/concerns` as a place to keep all your application's model *concerns*. Any file found within this directory will automatically be part of the application load path.

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.

9.10 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 class¹¹ 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.

Listing 9.5 Runtime Metaprogramming with `after_find`

```
1 class Account < ActiveRecord::Base
2   ...
3
4   protected
5
6   def after_find
7     singleton = class << self; self; end
8     singleton.class_eval(config)
9   end
10 end
```

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.

11. I don't expect this to make sense to you unless you are familiar with Ruby's singleton classes and have the ability to evaluate arbitrary strings of Ruby code at runtime. A good place to start is <http://yehudakatz.com/2009/11/15/metaprogramming-in-ruby-its-all-about-the-self/>

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:

```
1 validates_numericality_of :product_code, only_integer: true
2 validates_length_of      :product_code, is: 10
```

9.10.1 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.

9.10.2 Ruby and Domain-Specific Languages

My former colleague Jay Fields and I pioneered the mix of Ruby metaprogramming, Rails, and internal¹² 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 short¹³). When developing BNLs, you craft a domain-specific language that is not necessarily valid Ruby

12. The qualifier *internal* is used to differentiate a domain-specific language hosted entirely inside of a general-purpose language, such as Ruby, from one that is completely custom and requires its own parser implementation.

13. 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://blog.jayfields.com/2006/07/business-natural-language-material.html>

syntax but is close enough to be transformed easily into Ruby and executed at run-time, as shown in Listing 9.6.

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.

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 monkey patching 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.

9.11 Using Value Objects

In domain-driven design¹⁴ (DDD), there is a distinction 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.

14. <http://www.domaindrivendesign.org/>

Comparing two different Entity Objects for equality should always return false, even if all 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 `ActiveRecord::Base` but instead simply define a standard Ruby object. 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 the standard Ruby object provides a means to interact with those values in a more object-oriented way.

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:

```
1 class CreatePeople < ActiveRecord::Migration
2   def change
3     create_table :people do |t|
4       t.string :name
5       t.string :address_city
6       t.string :address_state
7     end
8   end
9 end
```

The Person model looks like this:

```
1 class Person < ActiveRecord::Base
2   def address
3     @address ||= Address.new(address_city, address_state)
4   end
5
6   def address=(address)
7     self[:address_city] = address.city
8     self[:address_state] = address.state
9
10    @address = address
11  end
12 end
```

We need a corresponding Address object, which looks like this:

```
1 class Address
2   attr_reader :city, :state
3
4   def initialize(city, state)
5     @city, @state = city, state
6   end
7
8   def ==(other_address)
9     city == other_address.city && state == other_address.state
10  end
11 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:0x007fcbfcce0188 @city="Brooklyn", @state="NY">
```

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

```
>> gary.address = Address.new("Brooklyn", "NY")
>> gary.address
=> #<Address:0x007fcbfa3b2e78 @city="Brooklyn", @state="NY">
```

9.11.1 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 on the parent object.

9.11.1.1 The Money Gem

A common approach to using Value Objects is in conjunction with the money gem.¹⁵

```
1 class Expense < ActiveRecord::Base
2   def cost
3     @cost ||= Money.new(cents || 0, currency || Money.default_currency)
4   end
```

15. <https://github.com/RubyMoney/money>

```
5
6   def cost=(cost)
7     self[:cents] = cost.cents
8     self[:currency] = cost.currency.to_s
9
10    cost
11  end
12 end
```

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

```
1 class CreateExpenses < ActiveRecord::Migration
2   def change
3     create_table :expenses do |t|
4       t.integer :cents
5       t.string :currency
6     end
7   end
8 end
```

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

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

9.12 Nonpersisted Models

In Rails 3, if one wanted to use a standard Ruby object with Action View helpers, such as `form_for`, the object had to “act” like an Active Record instance. This involved including/extending various Active Model module mixins and implementing the method `persisted?`. At a minimum, `ActiveModel::Conversion` should be included and `ActiveModel::Naming` extended. These two modules alone provide the object all the methods it needs for Rails to determine partial paths, routes, and naming. Optionally, extending `ActiveModel::Translation` adds internationalization support to your object, while including `ActiveModel::Validations` allows for validations to be defined. All modules are covered in detail in the Active Model API Reference.

To illustrate, let's assume we have a `Contact` class that has attributes for `name`, `email`, and `message`. The following implementation is Action Pack and Action View compatible in both Rails 3 and 4:

```

1 class Contact
2   extend ActiveRecord::Naming
3   extend ActiveRecord::Translation
4   include ActiveRecord::Conversion
5   include ActiveRecord::Validations
6
7   attr_accessor :name, :email, :message
8
9   validates :name, presence: true
10  validates :email,
11    format: { with: /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/ },
12    presence: true
13  validates :message, length: {maximum: 1000}, presence: true
14
15  def initialize(attributes = {})
16    attributes.each do |name, value|
17      send("#{name}=", value)
18    end
19  end
20
21  def persisted?
22    false
23  end
24 end

```

New to Rails 4 is the `ActiveModel::Model`, a module mixin that removes the drudgery of manually having to implement a compatible interface. It takes care of including/extending the modules mentioned earlier, defines an initializer to set all attributes on initialization, and sets `persisted?` to `false` by default. Using `ActiveModel::Model`, the `Contact` class can be implemented as follows:

```

1 class Contact
2   include ActiveRecord::Model
3
4   attr_accessor :name, :email, :message
5
6   validates :name, presence: true
7   validates :email,
8     format: { with: /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/ },

```

```

9     presence: true
10   validates :message, length: {maximum: 1000}, presence: true
11 end

```

9.13 PostgreSQL Enhancements

Out of all the supported databases available in Active Record, PostgreSQL received the most amount of attention during the development of Rails 4. In this section, we are going to look at the various additions made to the PostgreSQL database adapter.

9.13.1 Schemaless Data with **hstore**

The `hstore` data type from PostgreSQL allows for the storing of key/value pairs or simply a hash within a single column. In other words, if you are using PostgreSQL and Rails 4, you can now have schema-less data within your models.

To get started, first set up your PostgreSQL database to use the `hstore` extension via the `enable_extension` migration method:

```

1 class AddHstoreExtension < ActiveRecord::Migration
2   def change
3     enable_extension "hstore"
4   end
5 end

```

Next, add the `hstore` column type to a model. For the purpose of our examples, we will be using a `Photo` model with an `hstore` attribute `properties`.

```

1 class AddPropertiesToPhotos < ActiveRecord::Migration
2   change_table :photos do |t|
3     t.hstore :properties
4   end
5 end

```

With the `hstore` column `properties` set up, we are able to write a hash to the database:

```

1 photo = Photo.new
2 photo.properties # nil
3 photo.properties = { aperture: 'f/4.5', shutter_speed: '1/100 secs' }
4 photo.save && photo.reload
5 photo.properties # {:aperture=>"f/4.5", :shutter_speed=>"1/100 secs"}

```

Although this works well enough, Active Record does not keep track of any changes made to the `properties` attribute itself.

```

1 photo.properties[:taken] = Time.current
2 photo.properties

```

```

3 # {:aperture=>"f/4.5", :shutter_speed=>"1/100 secs",
4 # :taken=>Wed, 23 Oct 2013 16:03:35 UTC +00:00}
5
6 photo.save && photo.reload
7 photo.properties # {:aperture=>"f/4.5", :shutter_speed=>"1/100 secs"}

```

As with some other PostgreSQL column types, such as array and json, you must tell Active Record that a change has taken place via the `<attribute>_will_change!` method. However, a better solution is to use the Active Record `store_accessor` macro-style method to add read/write accessors to hstore values.

```

1 class Photo < ActiveRecord::Base
2   store_accessor :properties, :aperture, :shutter_speed
3 end

```

When we set new values to any of these accessors, Active Record is able to track the changes made to the underlying hash, eliminating the need to call the `<attribute>_will_change!` method. Like any accessor, they can have Active Model validations added to them and also can be used in forms.

```

1 photo = Photo.new
2 photo.aperture = "f/4.5"
3 photo.shutter_speed = "1/100 secs"
4 photo.properties # {"aperture"=>"f/4.5", "shutter_speed"=>"1/100 secs"}
5
6 photo.save && photo.reload
7
8 photo.properties # {"aperture"=>"f/4.5", "shutter_speed"=>"1/100 secs"}
9 photo.aperture = "f/1.4"
10
11 photo.save && photo.reload
12 photo.properties # {"aperture"=>"f/1.4", "shutter_speed"=>"1/100 secs"}

```

Be aware that when an hstore attribute is returned from PostgreSQL, all key/values will be strings.

9.13.1.1 Querying hstore

To query against an hstore value in Active Record, use SQL string conditions with the `where` query method. For the sake of clarity, here are a couple examples of various queries that can be made against an hstore column type:

```

1 # Nonindexed query to find all photos that have a key 'aperture' with a
2 # value of f/1.4
3 Photo.where("properties -> :key = :value", key: 'aperture', value: 'f/1.4')
4

```

```

5 # Indexed query to find all photos that have a key 'aperture' with a value
6 # of f/1.4
7 Photo.where("properties @> 'aperture=>f/1.4'")
8
9 # All photos that have a key 'aperture' in properties
10 Photo.where("properties ? :key", key: 'aperture')
11
12 # All photos that do not have a key 'aperture' in properties
13 Photo.where("not properties ? :key", key: 'aperture')
14
15 # All photos that contains all keys 'aperture' and 'shutter_speed'
16 Photo.where("properties ?& ARRAY[:keys]", keys: %w(aperture shutter_speed))
17
18 # All photos that contains any of the keys 'aperture' or 'shutter_speed'
19 Photo.where("properties ?| ARRAY[:keys]", keys: %w(aperture shutter_speed))

```

For more information on how to build hstore queries, you can consult the PostgreSQL documentation directly.¹⁶

9.13.1.2 GiST and GIN Indexes

If you are doing any queries on an hstore column type, be sure to add the appropriate index. When adding an index, you will have to decide to use either GIN or GiST index types. The distinguishing factor between the two index types is that GIN index lookups are three times faster than GiST indexes; however, they also take three times longer to build.

You can define either a GIN or GiST index using Active Record migrations by setting the index option `:using` to `:gin` or `:gist`, respectively.

```

add_index :photos, :properties, using: :gin
# or
add_index :photos, :properties, using: :gist

```

GIN and GiST indexes support queries with `@>`, `?`, `?&`, and `?|` operators.

9.13.2 Array Type

Another NoSQL-like column type supported by PostgreSQL and Rails 4 is `array`. This allows us to store a collection of a data type, such as strings, within the database record itself. For instance, assuming we had an `Article` model, we could store all the article's tags in an array attribute named `tags`. Since the tags are not stored in

16. <http://www.postgresql.org/docs/9.3/static/hstore.html>

another table, when Active Record retrieves an article from the database, it does so in a single query.

To declare a column as an array, pass `true` to the `:array` option for a column type such as `string`:

```
1 class AddTagsToArticles < ActiveRecord::Migration
2   def change
3     change_table :articles do |t|
4       t.string :tags, array: true
5     end
6   end
7 end
8 # ALTER TABLE "articles" ADD COLUMN "tags" character varying(255)[]
```

The array column type will also accept the option `:length` to limit the amount of items allowed in the array.

```
t.string :tags, array: true, length: 10
```

To set a default value for an array column, you must use the PostgreSQL array notation (`{value}`). Setting the default option to `{}` ensures that every row in the database will default to an empty array.

```
t.string :tags, array: true, default: '{}rails,ruby'
```

The migration in the previous code sample would create an array of strings that defaults every row in the database to have an array containing strings “rails” and “ruby.”

```
>> article = Article.create
(0.1ms) BEGIN
SQL (66.2ms) INSERT INTO "articles" ("created_at", "updated_at") VALUES
($1, $2) RETURNING "id" [{"created_at", Wed, 23 Oct 2013 15:03:12

>> article.tags
=> ["rails", "ruby"]
```

Note that Active Record does not track destructive or in-place changes to the `Array` instance.

```
1 article.tags.pop
2 article.tags # ["rails"]
3 article.save && article.reload
4 article.tags # ["rails", "ruby"]
```

To ensure changes are persisted, you must tell Active Record that the attribute has changed by calling `<attribute>_will_change!`.

```
1 article.tags.pop
2 article.tags # ["rails"]
3 article.tags_will_change!
4 article.save && article.reload
5 article.tags # ["rails"]
```

If the `pg_array_parser` gem is included in the application `Gemfile`, Rails will use it when parsing PostgreSQL's array representation. The gem includes a native C extension and JRuby support.

9.13.2.1 Searching in Arrays

If you wish to query against an array column using Active Record, you must use PostgreSQL's methods `ANY` and `ALL`. To demonstrate, given our previous example, using the `ANY` method, we could query for any articles that have the tag "rails":

```
Article.where("'rails' = ANY(tags)")
```

Alternatively, the `ALL` method searches for arrays where all values in the array equal the value specified.

```
Article.where("'rails' = ALL(tags)")
```

As with the `hstore` column type, if you are doing queries against an array column type, the column should be indexed with either `GiST` or `GIN`.

```
add_index :articles, :tags, using: 'gin'
```

9.13.3 Network Address Types

PostgreSQL comes with column types exclusively for IPv4, IPv6, and MAC addresses. IPv4 or IPv6 host address are represented with Active Record data types `inet` and `cidr`, where the former accepts values with nonzero bits to the right of the netmask. When Active Record retrieves `inet/cidr` data types from the database, it converts the values to `IPAddr` objects. MAC addresses are represented with the `macaddr` data type, which are represented as a string in Ruby.

To set a column as a network address in an Active Record migration, set the data type of the column to `inet`, `cidr`, or `macaddr`:

```
1 class CreateNetworkAddresses < ActiveRecord::Migration
2   def change
3     create_table :network_addresses do |t|
4       t.inet :inet_address
```

```

5      t.cidr :cidr_address
6      t.macaddr :mac_address
7  end
8 end
9 end

```

Setting an `inet` or `cidr` type to an invalid network address will result in an `IPAddr::InvalidAddressError` exception being raised. If an invalid MAC address is set, an error will occur at the database level resulting in an `ActiveRecord::StatementInvalid: PG::InvalidTextRepresentation` exception being raised.

```

>> address = NetworkAddress.new
=> #<NetworkAddress id: nil, inet_address: nil, ...>

```

```

>> address.inet_address = 'abc'
IPAddr::InvalidAddressError: invalid address

```

```

>> address.inet_address = "127.0.0.1"
=> "127.0.0.1"

```

```

>> address.inet_address
=> #<IPAddr: IPv4:127.0.0.1/255.255.255.255>

```

```

>> address.save && address.reload
=> #<NetworkAddress id: 1,
    inet_address: #<IPAddr: IPv4:127.0.0.1/255.255.255.255>, ...>

```

9.13.4 UUID Type

The `uuid` column type represents a universally unique identifier (UUID), a 128-bit value that is generated by an algorithm that makes it highly unlikely that the same value can be generated twice.

To set a column as a UUID in an Active Record migration, set the type of the column to `uuid`:

```
add_column :table_name, :unique_identifier, :uuid
```

When reading and writing to a UUID attribute, you will always be dealing with a Ruby string:

```
record.unique_identifier = 'a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11'
```

If an invalid UUID is set, an error will occur at the database level, resulting in an `ActiveRecord::StatementInvalid: PG::InvalidTextRepresentation` exception being raised.

9.13.5 Range Types

If you have ever needed to store a range of values, Active Record now supports PostgreSQL range types. These ranges can be created with both inclusive and exclusive bounds. The following range types are natively supported:

- `daterange`
- `int4range`
- `int8range`
- `numrange`
- `tsrange`
- `tstzrange`

To illustrate, consider a scheduling application that stores a date range representing the availability of a room.

```
1 class CreateRooms < ActiveRecord::Migration
2   def change
3     create_table :rooms do |t|
4       t.daterange :availability
5     end
6   end
7 end
8
9 room = Room.create(availability: Date.today..Float::INFINITY)
10 room.reload
11 room.availability # Tue, 22 Oct 2013...Infinity
12 room.availability.class # Range
```

Note that the `Range` class does not support exclusive lower bound. For more detailed information about the PostgreSQL range types, consult the official documentation.¹⁷

9.13.6 JSON Type

Introduced in PostgreSQL 9.2, the `json` column type adds the ability for PostgreSQL to store JSON structured data directly in the database. When an Active Record object has an attribute with the type of `json`, the encoding/decoding of the JSON itself is handled behind the scenes by `ActiveSupport::JSON`. This allows you to set the attribute to a hash or already encoded JSON string. If you attempt to set the JSON attribute to a string that cannot be decoded, a `JSON::ParserError` will be raised.

17. <http://www.postgresql.org/docs/9.3/static/rangetypes.html>

To set a column as JSON in an Active Record migration, set the data type of the column to `json`:

```
add_column :users, :preferences, :json
```

To demonstrate, let's play with the `preferences` attribute from the previous example in the console. To begin, I'll create a user with the color preference of blue.

```
>> user = User.create(preferences: { color: "blue" } )
(0.2ms) BEGIN
SQL (1.1ms) INSERT INTO "users" ("preferences") VALUES ($1) RETURNING
"id" [["preferences", {:color=>"blue"}]]
(0.4ms) COMMIT
=> #<User id: 1, preferences: {:color=>"blue"}>
```

Next up, let's verify when we retrieve the user from the database that the `preferences` attribute doesn't return a JSON string but a hash representation instead.

```
>> user.reload
User Load (10.7ms) SELECT "users".* FROM "users" WHERE "users"."id" = $1
LIMIT 1 [["id", 1]]
=> #<User id: 1, preferences: {"color"=>"blue"}>

>> user.preferences.class
=> Hash
```

It's important to note that like the array data type, Active Record does not track in place changes. This means that updating the existing hash does not persist the changes to the database. To ensure changes are persisted, you must call `<attribute>_will_change!` (`preferences_will_change!` in our previous example) or completely replace the object instance with a new value instead.

9.14 Conclusion

With this chapter we conclude our coverage of Active Record. Among other things, we examined how callbacks let us factor our code in a clean and object-oriented 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.

This page intentionally left blank

Index

Symbols

`-`, 674, 695, 764
`[]`, 149, 443, 631
```, 713  
`+`, 674, 695, 764  
`<<`, 199, 214, 215, 237  
`<=>`, 683, 725, 764, 775  
`=~`, 775  
`==`, 764

## A

`abstract_class=`, 166, 286  
`accepts_nested_attributes_for`, 362–365  
acronym, 707–708  
Action Controller  
  callbacks, 116–121  
    around, 119–120  
    classes 118  
    conditions, 120–121  
    halting, 121  
    inline method, 118  
    inheritance, 116–117

    ordering, 118–119  
    skipping, 120  
communication with view, 115  
controller specs, 601–604  
layouts, specifying, 111  
postbacks, 381  
rendering, 102–111  
standard RESTful actions, 69–73  
streaming content, 121–126  
variants, 126–127  
verify method, 117  
Action Dispatch, 99–102  
Action Mailer, 493–504  
  attachments, 497–498  
  callbacks, 499–500  
  custom email headers, 495  
  generating URLs inside messages, 498  
  handing inbound attachments, 500–501  
  HTML messages, 496  
  mailer layouts, 498–499

## Action Mailer (continued)

- models, 494–500
- multipart messages, 497
- preparing outbound messages, 494–496
- previews, 503–504
- raising delivery errors, 21
- receiving, 500–501
- sending, 499
- server configuration, 502
- SMTP, 493, 502
- testing with RSpec, 502–503

## Action View, 313–329

- conditional output, 316–317
- customizing validation error output, 337–338
- ERb. See ERb
- filename conventions, 314
- flash messages, 321–322
- Haml. See Haml
- Helpers. See Helpers
- instance variables, 318–320
- layouts, 314–315
- logging, 328
- partial\_counter variable, 328
- partials. See Partials
- rendering 327
- view specs, 604–605
- yielding content, 315–316

## Active Model, 625–649

- AttributeMethods module, 625–627
- Callbacks module, 627–629
- Conversion module, 629
- Dirty module, 629–631
- Errors class, 631–635
- ForbiddenAttributesError, 635
- Model, 635–636
- Naming module, 636–638

## SecurePassword, 638

- serialization, 638–640
- testing compatibility of custom classes with Lint::Tests, 635
- translation, 640–641
- Validations module, 641–649

## Active Record

- abstract base models, 286–287
- associations, 121, 195–240
  - :counter cache option, 207–208
  - :counter sql option, 200
  - :dependent option, 201, 208, 235
  - :finder sql option, 200
- AssociationProxy class, 239–240
- belongs\_to. See belongs\_to associations
- destroying records, 201
- extensions, 238–239
- foreign-key constraints, 292
- has\_and\_belongs\_to\_many. See has\_and\_belongs\_to\_many associations
- has\_many. See has\_many associations
- has\_many :through. See has\_many :through associations
- indexing, 509
- many-to-many relationships, 222–233
- one-to-many relationships, 196–204
- one-to-one relationships, 233–236
- polymorphic, 287–290
- size of, 202
- unique sets, 232–233
- unsaved objects, 236–238
- attributes, 133–137
- readonly, 149–150

- reloading, 142
- serialized, 136–137
- translation, 640–641
- typecasting, 142
- updating, 145–147
- Base class, 120
- basic object operations, 138–151
- calculation methods, 278–279
- callbacks, 268–278
- cloning, 142–143
- concurrency. *See* Database locking
- configuration, 171
- find\_by\_sql method, 143–144
- legacy naming schemes, 133
- migrations, 173–194
  - column type mappings, 182–183
  - creating, 173–187
  - magic timestamp columns, 186–187
  - schema.rb file, 189–190
  - sequencing, 174
- model specs, 599–601
- pattern, 129
- query caching, 144–145
- querying, 155–165
  - arel\_table, 165
  - exists, 160
  - extending, 160
  - from, 159–160
  - group, 160–161
  - having, 161
  - includes, 161–162
  - joins, 162
  - limit, 158–159
  - none, 162
  - offset, 158–159
  - order, 157–158
  - readonly, 163
  - references, 163
  - reorder, 163–164
  - reverse\_order, 164
  - select, 159
  - unique, 164
  - unscope, 164–165
  - where, 155–165
- RecordInvalid exception, 252–253
- RecordNotSaved exception, 200–201, 271
- records
  - deleting, 150–151
  - random ordering, 158
  - touching, 149
- scopes, 263–268
- session store, 14–15, 450–455
- STI (Single-Table Inheritance), 280–287
- translations, 413–417
- validations, 241–263
  - common options, 253–254
  - conditional validation, 255–256
  - contexts, 256
  - custom macros, 258–259
  - declarative, 242–253
  - errors, 241–242, 261
  - enforcing uniqueness of join
    - models, 250–251
  - reporting, 332–335
  - short-form, 256–258
  - skipping, 260–261
  - testing with Shoulda, 262
  - value objects, 299–302
- Active Support, 651–780
- active?, 170
- acts\_like?, 673–674, 682, 738, 750, 764
- adapter\_name, 170

- add, 632
  - add\_column, 181–182, 187, 197
  - added?, 632
  - add\_index, 179
  - add\_on\_blank, 632
  - add\_on\_empty, 632
  - add\_silencer, 12, 658
- advance, 674, 683, 764
- after, 580–581
  - after\_add, 214
  - after\_commit, 270
  - after\_create, 270
  - after\_destroy, 258, 262, 270, 274
  - after\_filter, 473–474
  - after\_find, 269, 274–275, 297
  - after\_fork, 539
  - after\_initialize, 137, 269, 274–276
  - after\_remove, 215
  - after\_rollback, 270
  - after\_save, 270
  - after\_touch, 270
  - after\_update, 270
  - after\_validation, 269–270, 643
- ago, 674, 683, 695, 737, 765
- Ajax, 91, 545–558
  - HTML fragments, 555–557
  - JSON, 553–555
  - JSONP, 557–558
  - Unobtrusive JavaScript (UJS), 547–550
- alias\_attribute, 626, 716–717
- alias\_method\_chain, 529, 717–718
- all\_day, 765
- all\_month, 765
- all\_quarter, 765
- all\_week, 765
- all\_year, 765
- allow\_forgery\_protection, 23
- and\_return, 593
- anonymous?, 718
- any?, 199
- append, 656
- arel\_table, 165
- around\_create, 270
- around\_save, 270
- Array, extensions, 651–657
- Array.wrap, 657
- as: association\_name, 215
- as\_json, 632–633, 639, 659–660, 682, 687, 697
- assert
  - assert\_difference, 761–762
  - assert\_equal, 584
- assert\_no\_difference, 762
- assert\_no\_match, 759
- assert\_not, 762
  - assert\_not\_empty, 760
  - assert\_not\_equal, 760
  - assert\_not\_in\_delta, 760
  - assert\_not\_in\_epsilon, 760
  - assert\_not\_includes, 760
  - assert\_not\_instance\_of, 760
  - assert\_not\_kind\_of, 760
  - assert\_not\_nil, 760
  - assert\_not\_operator, 760
  - assert\_not\_predicate, 760
  - assert\_not\_respond\_to, 760
  - assert\_not\_same, 760
- assert\_nothing\_raised, 760
- assert\_raise, 761
- assert\_valid\_keys, 701–702
- asset\_host, 26, 343–345
- Asset hosts, 343–345
- Asset Pipeline, 559–574
  - file serving, 572–573
  - fingerprinting, 571–572

- compression, 569, 573
- Data URL, 571
- helpers, 569–570
  - built-in SASS asset path helpers, 570–571
  - getting the URL of an asset file, 570
  - images, 570
- manifest files, 561–567
  - directives, 563–564
  - format handlers, 565–568
  - gemified assets, 564–565
  - index files, 565
  - search path, 564
- postprocessing, 568–569
  - custom compressor, 569
  - JavaScripts, 568
  - stylesheets, 568
- rake tasks, 573–674
- template engines, 566–567
- web server configuration, 572–573
- Assets, 25–26
- Assets debug mode, 22
- assigns, 318
- associate\_with, 688
- Asynchronous processing. *See* Background processing.
- at, 749, 776
  - at\_beginning\_of\_day, 674, 683, 765
  - at\_beginning\_of\_hour, 683, 765
  - at\_beginning\_of\_minute, 683, 765
  - at\_beginning\_of\_month, 674–675
  - at\_beginning\_of\_quarter, 675, 766
  - at\_beginning\_of\_week, 675, 766
  - at\_beginning\_of\_year, 675, 766
  - at\_end\_of\_day, 675, 683, 766
  - at\_end\_of\_hour, 683, 766
  - at\_end\_of\_minute, 683, 766
  - at\_end\_of\_month, 675, 766
  - at\_end\_of\_quarter, 675, 766
  - at\_end\_of\_week, 675, 766
  - at\_end\_of\_year, 675, 766
  - at\_midnight, 674, 683, 765
- Atom Feeds
  - autodetection, 338
  - atom feed method, 346–347
- atomic\_write, 698–699
- attr\_accessible, 481
- attr\_internal\_accessor, 718–719
- attr\_internal\_reader, 718
- attr\_internal\_writer, 719
- attr\_protected, 481
- attr\_readonly, 150
- attribute\_methods.rb, 626–627
  - attribute\_method\_affix, 626
  - attribute\_method\_prefix, 626
  - attribute\_method\_suffix, 626
- attributes, 141
- attributes=, 141
- audio\_path, 341
- audio\_tag, 341
- authenticate\_user!, 462
- Authentication
  - Active Resource, 467
  - client-side certificates, 490
  - HTTP basic, 467
  - HTTP digest, 468
- auto\_discovery\_link\_tag, 338–339
- autoclose, 446
- autoload, 691
  - autoload\_at, 692
  - autoload\_module!, 689
  - autoload\_once\_paths, 687
  - autoload\_paths, 687–688
  - autoload\_under, 692
  - autoloadable\_module?, 689

- autoload (continued)
  - autoloader?, 689
  - autoloads, 692
- average, 199, 278–279
- await, 673

## B

- background processing, 527–543
- backtrace\_cleaner, 12, 657
- base\_path, 318
- be\_an\_error, 530
- be\_routable, 604
- before, 580–581
  - before\_action, 116
  - before\_add, 215–216
  - before\_create, 270, 628–629
  - before\_destroy, 151, 216, 268–270, 273–274, 276
  - before\_first\_fork, 539
  - before\_fork, 539
  - before\_perform, 539
  - before\_remove, 216
  - before\_save, 272–273, 666–667
  - before\_update, 270
  - before\_validation, 269–271
- begin\_db\_transaction, 167
- beginning\_of\_\*
  - beginning\_of\_day, 674, 683, 765
  - beginning\_of\_hour, 683, 765
  - beginning\_of\_minute, 683, 765
  - beginning\_of\_month, 674–675
  - beginning\_of\_quarter, 675, 766
  - beginning\_of\_week, 676, 767
- belongs\_to associations, 149, 180, 196, 205–214
  - building and creating related objects, 206
  - options, 206–211

- polymorphic, 197
- reloading, 205
- scopes, 211–214
- touch, 210
- benchmarking, 120, 213, 219, 221, 658–659
- Better Errors gem, 782–783
- binary data storage, 185
- blank?, 633, 657, 698, 704, 729, 730, 739, 749, 778
- breadcrumbs, 424–425
- build, 199–200
- build\_association, 206
- Builder::XmlMarkup class, 617, 620–622
- Bundler, 2–7
- button\_tag, 379
- button\_to, 418, 548–549
- by, 510
- byte / bytes, 731

## C

- cache, 510
  - cache\_classes=, 18
  - cache\_directory, 507
  - cache\_if, 516
  - cache\_key, 513, 517, 525, 637, 660
  - cache\_path, 520
  - cache\_store=, 452–453, 519–520, 522
  - cache\_sweeper, 518
  - cache\_unless, 516
  - caches\_action, 506, 508, 510
  - caches\_page, 506–507
- Caching
  - :counter\_cache, 195–196
  - action caching, 508–509
  - avoiding extra database activity

- during, 518–519, 521–522
- CacheHelper module, 347
- conditional caching, 516
- controlling web caches and proxies, 523–524
- disabling in development mode, 20
- ETags, 524–526
- expiration, 516–518
- fetch, 522–523
- fragment caching, 509–516
- logging, 519
- new caches, 522
- page caching, 506–507
- query caching, 145
- storage, 519–521
- Store class, 592–597
- sweeping, 508, 517–518
- view caching, 505–521
- calculate, 200
- Callbacks
  - Action Controller, 116–121
  - Action Mailer, 499–500
  - Active Model module, 627–629
  - Active Record, 268–278
    - has\_many associations, 214
    - new in Rails 4, 499–500
- callbacks.rb, 628–629
- camelize, 707–708, 752–753
- capitalize, 725
- capture, 348, 713
- CAS, 465
- cattr\_accessor, 670
- cattr\_reader, 671
- cattr\_writer, 671
- CDATA, 393, 446–447
- change, 178, 584–585, 676, 684, 767
  - change\_default, 178–179
  - change\_table, 178, 197
  - changed, 630
  - changed\_attributes, 631
  - changed?, 630
  - changes, 631
- chars proxy, 756–757
- check\_box, 367
- check\_box\_tag, 379
- Class
  - automatic reloading, 18–20
  - extensions, 668–671
  - Rails class loader, 18–19
- class << self, 297
- class\_attribute, 668–670
- class\_eval, 297, 299, 714
- classify, 753
- cleanup, 661
- clear, 200, 261, 661, 689, 708
- clear\_query\_cache, 145
- clone, 739
- collection\_check\_boxes, 373–374
- collection\_radio\_buttons, 374–375
- collection\_select, 372, 374
- CollectionProxy, 239–240
- color\_field, 367
- color\_field\_tag, 379
- column, 179
- commit\_db\_transaction, 168
- compact, 699
- compact!, 699
- compiler\_class, 447
- compose, 725
- concat, 395
- concern, 719–720
- Concern module, 671–672
- concerning, 719
- Concurrency. See Database Locking
- config, 673
- config\_accessor, 673

- Configurable module, 673
- configure, 673
- consider\_all\_requests\_local=, 20
- console, 17
- const\_missing, 688, 690, 724
- constant\_watch\_stack, 688
- constantize, 689, 753
- content\_for, 315–316, 348–349
- content\_for?, 349
- content\_tag, 394
- content\_tag\_for, 390–391
- context, 578
- controller, 318–319
- Controllers. *See* Action Controller
- Convention over configuration, 103, 129, 132, 171, 401
- Cookies, 319, 455–457
  - :secure option, 456
  - reading and writing, 455–456
  - session store, 14–15, 453–455
  - signing, 456
- copy\_instance\_variables\_from, 718
- count, 200, 279, 633
- count\_by\_sql, 144
- Country Select gem, 783
- create, 199, 200–201
  - create\_association, 206
  - create\_join\_table, 178
  - create\_table, 176–178, 223
  - create!, 200–201
  - created\_at, 186
  - created\_on, 186
- CRUD (Create Read Update Delete), 138–151
  - routing and, 66–69
- CSS
  - linking stylesheets to template, 340
  - relation to Haml, 446

- sanitizing, 393
- Currency
  - formatting, 385
  - Money gem, 301–302
- current
  - current\_cycle, 396
  - current\_page, 419
  - current\_page?, 419
- cycle, 396

## D

- dasherize, 12, 614, 654, 753
- Data migration, 187–189
- Databases
  - configuring, 26–27, 171
  - connecting to multiple, 165–166
  - foreign-key constraints, 292
  - locking, 151–155
    - considerations, 154–155
    - optimistic, 152–154
    - pessimistic, 154
  - migrations. *See* ActiveRecord, Migrations.
  - schemas, 16–17, 152, 171–172
  - seeding, 190–191
  - using directly, 167–169
- Date, extensions, 673–682
- date\_field, 367
- date\_field\_tag, 380
- DATE\_FORMATS hash constant, 770
  - DateHelper module, 349–356
- Date input tags, 367, 380
- date\_select, 350
- datetime
  - datetime\_field, 367–368
  - datetime\_field\_tag, 380
  - datetime\_local\_field, 368
  - datetime\_local\_field\_tag, 380

- datetime\_select, 351, 416
- DateTime, extensions, 682–687
- day / days, 737
  - days\_ago, 677, 767
  - days\_in\_month, 767
  - days\_since, 677, 767
  - days\_to\_week\_start, 677, 767
- Debugger gem, 783
- Decent Exposure gem, 105, 317–318
- decimal precision, 184–185
- decode, 712
- decompose, 725
- decrement, 271, 662
- decrypt\_and\_verify, 715
- deep\_merge, 700
- deep\_merge!, 700
- deep\_stringify\_keys, 702
- deep\_symbolize\_keys, 702
- deep\_transform\_keys, 702
- default\_locale, 402
- default\_scope, 266
- default\_timezone=, 171
- define\_attribute\_method, 627
- define\_attribute\_methods, 627
- define\_callbacks, 665–667
- define\_model\_callbacks, 628–629
- delay, 534
- delay\_for, 534
- delay\_until, 534
- Delayed Job gem, 528–530
- delegate, 720–721
- delete, 168, 201, 271, 633, 662
- delete\_all, 199–201, 271
- delete\_matched, 662
- demodulize, 753
- depend\_on, 689
- deprecate\_methods, 693, 722
- Deprecation, 21, 693–694, 722, 745

- Deprecation.behavior, 693
- descendants, 694
- describe, 578
- destroy\_all, 201
- destroyed?, 151
- Devise gem, 459–466
- direct\_descendants, 694
- disconnect!, 170
- distance\_of\_time\_in\_words, 355–356, 416
- distance\_of\_time\_in\_words\_to\_now, 356
- distinct, 222
- div\_for, 327–328, 391
- does\_not\_match?, 588
- dom\_id, 426
- Domain-Specific Languages, 131, 298
- downcase, 726
- Draper gem, 783–784
- drop\_table, 192
- duplicable?, 739–740
- Duration class, 695–696

## E

- each, 633
- Eager load, 20
  - eager\_autoload, 692
  - eager\_load!, 692
- element, 637
- Email. See Action Mailer
- email\_field, 368
- email\_field\_tag, 380
- empty?, 201, 633
- enable\_warnings, 713
- encode, 712
- encode\_json, 730, 749
- encode64, 779
- encrypt\_and\_sign, 715
- end\_of\_\*

- end\_of\_\* (continued)
  - end\_of\_day, 675, 683, 766
  - end\_of\_hour, 683, 766
  - end\_of\_minute, 683, 766
  - end\_of\_month, 675, 766
  - end\_of\_quarter, 675, 766
  - end\_of\_week, 677, 767
  - end\_of\_year, 675, 766
- ends\_with?, 757
- enqueue, 529–530, 532–533
- Enumerable, extensions, 696–697
- ERb, 313, 697–698
  - asset pipeline, 566, 570
  - Devise, 463
  - email templates, 496–497
  - HamI versus, 433–434, 439, 463.
    - See also HamI
  - rendering inline template code, 106
- error\_message\_on, 332–333
- error\_messages, 332–334
- error\_messages\_for, 333–334
- errors, 261, 642
- escape\_attrs, 447
- escape\_html, 447
- escape\_javascript, 385
- escape\_once, 394
- establish\_connection, 164–166
- ETags, 524–525
- exabyte / exabytes2, 731
- except, 700–701
- except!, 701
- exception\_handler, 418
- excerpt, 397
- Excerpting text, 397
- exclude?, 696, 750
- execute, 168
- exist?, 523, 662
- exists?, 160

- expect, 583–584
- expire
  - expire\_action, 516–517
  - expire\_fragment, 516–517
  - expire\_page, 516–517
  - expires\_in, 520
  - expires\_now, 524
- explicitly\_unloadable\_constants, 688
- extending, 160, 219
- extract!, 703
- extract\_options!, 655

## F

- Facebook Open Graph meta tags, 316
- fallbacks=, 25
- favicon\_link\_tag, 339
- favicon.ico file, 339
- fetch, 522–523, 662–664
- field\_set\_tag, 380
- fields\_for, 362–365
- fifth, 652
- file\_field, 368
- file\_field\_tag, 380–381
- Files
  - extensions by Active Support, 698–699
  - reporting sizes to users, 385–389
  - upload field, 368, 380–381
- find
  - find\_by\_sql, 143–144
  - find\_tzinfo, 776
  - find\_zone, 771
  - find\_zone!, 772
- Firebug, 546
- first, 202, 749
- flash, 320–321
- flash.now, 321
- floats, 186, 659
- flush, 759

- font\_path, 341
- foreign\_key, 208, 217, 754
- form, 335–336
  - form\_for, 92, 358, 549
  - form\_tag, 381–382, 549
- format, 447
- formatted\_offset, 685, 770, 776
- Forms, 357–371
  - accepts\_nested\_attributes\_for method, 362–363
  - automatic view creation, 335–337
  - button\_to helper method, 418, 548–549
  - helper methods, 357–371
  - input, 366–371
- fortnight / fortnights, 737
- forty\_two, 652
- fourth, 652
- fragment\_exist?, 518–519
- freeze, 763
- fresh\_when, 525
- Friends gem, 786–788
- from, 159–160, 651, 749
  - from\_json, 639
  - from\_now, 695, 738
  - from\_xml, 622–624, 640, 699–700
- full\_messages, 633
- full\_messages\_for, 633
- future?, 677–678, 684, 767

## G

- Gemfile, 3–8
  - dependencies, 8
  - essential, 782–789
  - installing, 5–7
  - loading, 4–5
  - locking, 7
  - packaging 7–9

- generate, 716
  - generate\_key, 665, 714
  - generate\_message, 633–634
- generated\_attribute\_method, 627
- Geocoding, 272–273
- gigabyte / gigabytes, 731
- grapheme\_length, 726
- group, 160–161, 219
- grouped\_collection\_select, 372
- Gzip, 705

## H

- Haml, 433–448
  - attributes, 434–436
    - boolean, 436
    - classes and IDs, 436–438
    - data, 435–436
    - empty tags, 439
    - implicit divs, 438–439
  - comments, 440–441
  - configuration, 446–448
  - content, 445–446
  - creating elements, 434
  - doctype, 440
  - escaping, 442–443, 447
  - filters, 444–445
  - helpers, 443–444
  - HTML and, 440–441, 442, 447
  - interpolation, 442
  - multiline declarations, 443
- handle\_asynchronously, 529
- has\_and\_belongs\_to\_many associations, 222–226
  - bidirectional, 224–225
  - custom SQL, 211–213
  - extra columns, 225
  - real join models and habtm, 225–226
  - self-referential, 223–224

- has\_key?, 634
- has\_many associations, 214–222
  - :conditions option, 251
  - :include option, 213
  - callbacks, 214
- has\_many :through associations, 226–230
  - aggregating, 228–229
  - join models, 226–227, 229–230
  - options, 230–233
  - usage, 228
  - validations and, 229–230
- has\_one associations, 233–236
  - :as, 235
  - :class name option, 235
  - :dependent, 235
  - options, 235
  - scopes, 236
  - together with has\_many, 235
- has\_secure\_password, 466–467
- Hash, extensions, 699–704
  - Hash.from\_trusted\_xml, 699
  - Hash.from\_xml, 699–700
- HashWithIndifferentAccess class, 705
- having, 161, 219
- Helper methods
  - breadcrumbs helper, 424–425
  - helper specs, 605
  - photo\_for helper, 423–424
  - Title helper, 422–423
  - writing your own View helpers, 422–425
- helper\_method, 91, 468
- hidden\_field, 368
- hidden\_field\_tag, 382
- hide\_action, 116
- highlight, 397
- history, 688
- hook!, 689
- hour / hours, 738
- hstore data type, 304–305
- HTML
  - escaping, 442, 447, 485–486, 697–698
  - sanitizing, 487
  - tags
    - a, 419–421
    - audio, 341
    - empty, 439
    - form. See Forms.
    - image, 341–342
    - label, 368–369
    - option, 378–379
    - password, 383
    - script, 385
    - select, 383
    - submit, 383–384
    - video, 342
- html\_escape, 697–698
- html\_escape\_once(s), 698
- html\_safe, 757
- HTTP
  - foundation of REST, 62–64
  - role in routing, 38, 41, 54
  - stateless, 449
  - status codes, 109–111
  - verbs (GET, POST, etc.), 68–71, 74, 419, 549
- human, 637, 708
- human\_attribute\_name, 413–414, 417, 641
- human\_name, 419
- humanize, 754
- hyphenate\_data\_attrs, 447
- I**
  - i18n\_key, 637
  - i18n\_scope, 641

- ids, 202, 279
  - Image tags, 341–342
  - image\_path, 341–342
  - image\_submit\_tag, 382
  - image\_tag, 341–342
  - in, 679, 684, 696, 769
    - in?, 740
    - in\_groups, 655
    - in\_groups\_of, 656
    - in\_milliseconds, 738
    - in\_time\_zone, 682, 686, 758, 772
  - include, 202, 634, 740, 746–747
  - included, 295–296
  - includes, 161–162, 219–220
  - increment, 271, 664
  - increment\_counter, 271
  - indent, 752
  - indent!, 752
  - index, 179
  - index\_by, 696
  - inflections, 709
  - inherited, 694
  - initialize, 665, 714–716, 776
  - Initializers, 11–15
    - backtrace silencers.rb, 12–13
    - file parameter logging, 12
    - inflections.rb, 12–13, 705
    - mime\_types.rb, 13–14
    - session store.rb, 14–15, 455
    - wrap parameters, 15
  - input, 336–337
  - inquiry, 756
  - insert, 168
  - insert\_after, 98
  - insert\_before, 98
  - inspect, 695–696
  - instance\_eval, 118, 255, 269
  - instance\_values, 740
  - instance\_variable\_names, 740–741
  - instrument, 729–730
  - Integer, extensions, 711–712
  - Internationalization (I18n), 399–418
    - default locale, 11
    - exception handling, 417–418
    - i18n\_key, 637
    - i18n\_scope, 641
    - interpolation, 442
    - locale files, 409–410
    - methods, 400–401, 416–417
    - setting user locales, 405–406
    - setup, 401–402
    - storing custom translations, 413–316
  - invalid?, 642
  - inverse\_of, 208, 217–218
  - irregular, 709
  - is\_utf8?, 757
  - it, 581
- J**
- JavaScript
- Ajax and, 557–558
  - escaping, 385
  - helpers, 339–340, 385
  - postprocessing, 658
  - Unobtrusive JavaScript (UJS), 547–550
- javascript\_include\_tag, 339–340, 559, 569
  - javascript\_path, 340
  - javascript\_tag, 385
  - joins, 162
  - jQuery framework, 547, 550, 552, 557–558. See also Turbolinks
  - JSON, 64, 87, 712
    - :json, 107
    - Ajax and, 553–555

## JSON (continued)

- as\_json, 632–633
  - format segments, 46–47
  - output escaping, 487, 698
  - PostgreSQL column type, 310–311
  - Redis database, 532
  - serializers, 639
  - strings, 659–660, 682, 687, 749, 763
  - variants, 126
  - wrap parameters, 15
- json\_escape, 698
- JSONP, 107, 557–558

**K**

- Kernel, extensions, 712–715
- Kaminari gem, 784–785
- keys, 634
- kilobyte / kilobytes, 731
- kind, 649

**L**

- l, 401
- label, 368–369
- label\_tag, 382
- last, 202, 750
- last\_month / prev\_month, 678, 768
- last\_quarter / prev\_quarter, 678, 768
- last\_week, 678, 768
- last\_year / prev\_year, 678, 768
- LDAP, 465
- length, 202
- let, 578–580
- let!, 580
- limit, 158–159, 221, 726
- link\_to, 419–420, 549–550
- link\_to helper methods, 549–550
- link\_to\_if, 420
- link\_to\_unless, 420
- link\_to\_unless\_current, 420–421

list\_of, 444

load, 742

load?, 689

load\_file, 690

load\_missing\_constant, 690

load\_once\_path?, 690

loadable\_constants\_for\_path, 690

loaded, 688

local, 776

local\_assigns, 326–327

local\_constants, 722

local\_to\_utc, 776

Locale files, 409–410

localize, 401, 407

lock!, 154

log\_activity, 688

log\_level, 16

logger, 321

Logging, 29–35

backtrace silencing, 12–13, 657–658

colorization, 34

level override, 16

levels, 29–30

log file analysis, 32–35

Logger, extensions, 714–715

Syslog, 35

tagged, 32

lookup\_ancestors, 641

lookup\_store, 522

**M**

mail\_to, 421

many?, 199, 696–697

mark\_for\_destruction, 237–238

mark\_for\_unload, 690

match, 40–43

matches?, 59–60, 588–589

mattr\_accessor, 719

mattr\_reader, 719

- mattr\_writer, 719
- maximum, 203, 279
- mb\_chars, 724–727, 756–757
- mechanism, 688
- megabyte / megabytes, 731
- Memcached, session store, 452–453
- MessageEncryptor class, 715
- MessageVerifier class, 715–716
- method\_missing, 726
- middle\_of\_day / noon, 678, 684, 768
- Middleware (Rack), 96–98
- midnight, 683, 674, 683, 765
- MIME types, 13–14
- mime\_type, 447
- minimum, 203, 279
- minute / minutes, 738
- mock, 520
  - mock\_model, 605
  - mock\_with, 598
- model\_name, 638
- Module, extensions, 716–724
- monday, 678, 768
- MongoDB, 463, 528, 530
- month / months, 738
  - month\_field, 369
  - month\_field\_tag, 382
  - months\_ago, 678, 768
  - months\_since, 678, 768
- ms, 658
- multiline?, 747
- multiple\_of?, 712
- mute, 664
- MVC (Model-View-Controller), 2, 15, 37, 95, 106, 311

## N

- name\_path, 54–55
- name\_url, 54–55
- Named scopes. See Active Record, scopes

- namespace, 59
- Nested Form Fields gem, 785–786
- new, 91, 203
  - new\_constants\_in, 690
  - new\_record?, 138, 202, 236
- next\_\*
  - next\_month, 678, 768
  - next\_quarter, 678, 768
  - next\_week, 679, 768
  - next\_year, 679, 769
- Nonces, 454
- none, 162
- normalize, 726, 728
- Notifications, 729
- now, 777
- nsec, 685
- number\_field, 369
- number\_field\_tag, 382–383
- number\_to\_currency, 385–386, 417
- number\_to\_human\_size, 386–388
- number\_to\_percentage, 388
- number\_to\_phone, 388–389
- number\_with\_delimiter, 389, 417
- number\_with\_precision, 389–390, 417
- Numbers
  - conversions, 385–390
  - delimiters, 386–390, 731–736
  - extensions to Numeric class, 738–743

## O

- Object, extensions, 738–743
- object\_id, 205
- offset, 158–159, 222
- OpenSSL Digests, 665, 714–715
- option\_groups\_from\_collection\_for\_
  - select, 372, 375–376
- options, 664
- options\_for\_select, 376–378
- options\_from\_collection\_for\_select, 378

order, 222, 236  
 ordinal, 711–712  
 ordinalize, 712  
 overlaps?, 747

## P

page\_class, 444  
 param\_key, 637  
 parameterize, 56, 710, 754  
 params, 322  
 params hash, 322  
 parent, 722  
 parent\_name, 722  
 parents, 722–723  
 parse, 777  
 parser\_class, 447  
 partial\_counter, 328–329  
 Partial, 105–106, 322–329  
   passing variables to, 325–327  
   rendering collections, 327–328  
   rendering objects, 327  
   reuse, 324  
   shared, 324–325  
   wrapping and generalizing, 425–426  
 password\_field, 370  
 password\_field\_tag, 383  
 past?, 679, 684  
 pending, 582–583  
 perform\_caching=, 20, 22, 24, 509  
 period\_for\_local, 777  
 period\_for\_utf, 777  
 persisted?, 138, 302–303, 335, 629  
 petabyte / petabytes, 731  
 photo\_for, 423–424  
 pluck, 203, 279  
 Plugins, xlvii, 345–346, 540  
 plural, 637, 706  
 Pluralization

  i18n, 399–418  
   Inflections class, 705–711  
   Inflector class, 12  
 pluralize, 398, 754  
 pluralize\_table\_names=, 133  
 prepend, 657  
   prepend\_after\_filter, 118–119  
   prepend\_before\_filter, 118–119  
 presence, 739  
 present?, 739  
 preserve, 448  
 previous\_changes, 631  
 primary\_key, 210, 218  
 primary\_key\_prefix\_type=, 133  
 Prototype framework, 547  
 provide, 349  
 proxy\_owner, 240  
 proxy\_reflection, 240  
 proxy\_target, 240  
 Pry gems, 786–788  
 published\_prior\_to, 240  
 Pundit, 469–476  
   creating a policy, 471–472  
   controller integration, 472  
   policy scopes, 473–474  
   strong parameters, 474–475  
   testing policies, 475–476

## Q

qualified\_const\_defined?, 690, 723  
 qualified\_const\_get, 723  
 qualified\_const\_set, 723  
 qualified\_name\_for, 691  
 quietly, 713

## R

Rack, 96–98, 90–91  
   Rack::Sendfile middleware, 124

- RACK\_ENV variable, 1
- routes as Rack endpoints, 48
- radio\_button, 370
- radio\_button\_tag, 383
- Railcasts, 790
- Rails
  - class loader and reloading, 17, 615–619
  - configurations, 1–35
  - development of, xl, xliii–xliv
  - environments, 1–35, 781–782
  - essentials, 781–790
  - lib directory, 20
  - root directory, 4
  - runner, 541–543
  - scaffolding, 147, 174, 333, 434
  - screencasts, 789–790
  - settings, 9–18
    - application.rb file, 9–11
    - autoload\_paths, 16, 687, 689–690, 692
    - boot.rb file, 9
    - cherry-picking frameworks used, 10
    - custom environments, 23
    - development mode, 18–20
    - environment.rb file, 9, 502, 557, 599
    - generator defaults, 11
    - initializers. See Initializers
    - production mode, 23–26
    - test mode, 22–23
- Rails Admin gem, 787–788
- RAILS\_ENV variable, 1
- Railtie, 745–746
- raise\_delivery\_errors=, 21, 25
- raise\_error, 584–585
- Rake tasks
  - asset pipeline and, 573–574
  - database-related, 171, 191–194
  - listing routes, 53
  - Rack filters, 96–97
  - Rails log files, 30
  - Resque, 540–541
  - Rspec Rails gem, 596
  - Spring application preloader, 17
- Random ordering (of records), 157–158
- Range, extensions, 746–747
- range\_field, 370
- range\_field\_tag, 383
- raw, 390
- raw\_connection, 170
- reachable?, 723–724
- read, 664
- read\_attribute, 135–141
- read\_multi, 523–524, 664–665
- readable\_inspect, 681, 685
- readonly, 163, 213, 222, 236
- readonly\_attributes, 150
- reconnect!, 170
- record\_timestamps=, 187
- RecordNotFound exception, 139
- redefine\_method, 724
- redirect\_to, 54
- reference / references, 163, 180, 691
- Regexp, extensions, 747
- register, 14
  - register\_alias, 14
  - register\_javascript\_expansion, 345–346
  - register\_stylesheet\_expansion, 345–346
- release, 673
- reload, 240
- reload!, 102, 130
- remove, 180–181, 751
  - remove\_belongs\_to, 180–181

- remove (continued)
  - remove\_column, 187–189
  - remove\_constant, 691
  - remove\_filters, 658
  - remove\_index, 180
  - remove\_possible\_method, 724
  - remove\_references, 180–181
  - remove\_silencers!, 12, 658
  - remove\_timestamps, 181
  - remove\_unloadable\_constants!, 691
  - remove\_whitespace, 448
- remove!, 751
- rename, 181
- Rendering views, 102–111
  - another actions's template, 93, 102–104
  - explicit, 103, 104
  - implicit, 102–103
  - inline templates, 106
  - JSON, 107
  - nothing, 108
  - options, 108–111
  - partials, *see* *Partials*.
  - render\_views method, 603
  - text, 106
  - XML, 107
- reorder, 163–164
- reorder\_characters, 728
- replace, 203
- reply\_to, 495, 502
- Request handling
  - in routing, 99
  - redirecting, 111–115
- request, 322
- require, 18, 742
  - require\_dependency, 742
  - require\_or\_load, 689, 691, 742
- Rescuable module, 748
- rescue\_from, 748
- reset, 240
  - reset\_callbacks, 668
  - reset\_counters, 208
  - reset\_cycle, 398
  - reset\_sequence!, 168
- respond\_to, 322
- Resque gem, 537–541
- REST and RESTful design, 63–93
  - action set, 88–92
  - collection routes, 81–82
  - controller-only resources, 83–86
  - forms, 359
  - HTTP verbs, 68–69
  - member routes, 80–81
  - nested resources, 74–78
  - resources and representations, 64–65
  - routes, 37, 66–69
  - singular resource routes, 73
  - standard controller actions, 61–64
- reverse, 726
  - reverse\_merge, 703
  - reverse\_merge!, 703
  - reverse\_order, 164
  - reverse\_update, 703
- reversible, 175
- revert, 181
- rollback\_db\_transaction, 168
- route\_key, 638
- route\_to, 604
- Routing, 37–61
  - :id field, 44
  - concerns, 78–79
  - constraining by request method, 41
  - formats, 46–48
  - globbing, 51–52
  - listing, 60–61
  - match method, 40–43

- name\_path versus name\_url, 54–55
  - named, 52–57, 67–68
  - rack endpoints, 48
  - redirecting, 45–46
  - RESTful routes, 66–69
    - :format parameter, 86
    - collection, 79–82
    - controller mappings, 75
    - member, 81–81
    - nested, 74–78
    - singular and plural, 71–73
  - root routes, 50–51
  - routes.rb file, 39–40
  - scopes, 57–60
  - segment keys, 43–44, 49–50
  - RPX authentication, 465
  - RSpec, 575–610
    - custom expectation matchers, 588–591
    - fluent chaining, 590–591
    - generator settings, 11
    - grouping related examples, 578
    - let methods, 578–580
    - mocking and stubbing, 592–595
    - pending, 582–583
    - predicate matchers, 587–591
    - running specs, 595–596
    - shared behaviors, 591
    - shared context, 592
    - spec\_helper.rb file, 596–597, 607–608
    - subjects, 586–587
    - testing email, 502–503
    - tools, 609–610
  - RSS autodetection, 338–339
  - Ruby
    - \$LOAD\_PATH, 19
    - hashes, 375, 386, 450, 456, 657
    - macro-style methods, 131–133
    - Marshal API, 450
    - modules for reusing common behavior, 292–296
    - Ruby Toolbox, 789
  - RubyGems, xlvii
    - Bundler, 2–9
    - dependencies, 8
    - Git repository, loading directly from, 4–5
    - installing, 5–7
    - packaging, 7–9
- S**
- safe\_constantize, 691, 754
  - safe\_join, 390
  - sanitize, 393
  - sanitize\_css, 393
  - save, 147
  - save!, 147, 271
  - schema\_format, 16, 171
  - scope, 57
  - Scopes. See Active Record, scopes
  - search\_field, 370
  - search\_field\_tag, 383
  - search\_for\_file, 691
  - second / seconds, 652, 738
    - seconds\_since\_midnight, 684, 769
    - seconds\_to\_utc\_offset, 777
    - seconds\_until\_end\_of\_day, 684, 769
  - secret\_key\_base, 27
  - Security
    - cross-site request forgery (XSRF), 360, 487–490
    - cross-site scripting (XSS), 483–484
    - fixation attacks, 490–491
    - HTML, 485–487
    - log masking, 479–480
    - model mass-assignment attributes protection, 481–483

## Security (continued)

- password management, 477–479
- replay attacks, 454–455
- secrets, 491–492
- secure sockets layer. *See* SSL
- SQL injection, 143–144
- token handling, 489–490
- select, 159, 203–204, 213, 222, 372
  - select\_all, 168
  - select\_date, 351–352
  - select\_datetime, 352–353
  - select\_day, 352
  - select\_hour, 352
  - select\_minute, 352–353
  - select\_month, 353–354, 416
  - select\_one, 168–169
  - select\_second, 353–354
  - select\_tag, 383
  - select\_time, 354
  - select\_value, 169
  - select\_values, 169
  - select\_year, 354
- send\_data, 123–124
- send\_file, 124–126
- serializable\_hash, 639
- serve\_static\_assets=, 22
- Session Management, 449–456
  - cleaning old sessions, 450, 452, 455
  - RESTful storage considerations, 85
  - turning off sessions, 451
- session, 322
- set, 634
- set\_callback, 668
- Settings, 9–18
- should, 583–584
- should\_not, 583–584
- show\_exceptions=, 22
- Sidekiq, 531–537

- delayed Action Mailer, 533–534
- error handling, 536
- monitoring, 536–537
- running, 534–536
- scheduled jobs, 533
- workers, 523–533
- silence, 665, 715
  - silence!, 665
  - silence\_stream, 713
  - silence\_warnings, 713
- simple\_format, 398
- Simple Form gem, 788
- since, 679, 684, 696, 769
- singleton\_class, 714
- singular, 638, 709
- singular\_route\_key, 638
- singularize, 755
- site=, 204
- size, 204, 634
- skip\_callback, 668
- slice, 703–704
- slice!, 704, 726
- smtp\_settings, 502
- SOAP, 64
- specify, 581–582
- Specjour, 610
- split, 656, 727
- Spring application preloader, 17–18
- squish, 751
- SSL
  - certificates for asset hosts, 345
  - OpenSSL digests, 665, 714–715
  - serving protected assets, 345
- stale?, 525–526
- starts\_with?, 757
- State Machine gem, 788–789
- Static content, 51, 400, 506
- store\_full\_sti\_class=, 171

store\_translations, 412, 711

Streaming, 121–126

String

    extensions, 748–758

    usage versus symbols, 141

stringify\_keys, 702

StringInquirer class, 758

strip\_heredoc, 757

strip\_links, 393

strip\_tags, 393

stylesheet\_link\_tag, 340

stylesheet\_path, 340

subclasses, 671

submit, 370

submit\_tag, 383–384

submit\_to\_remote, 361

sum, 204, 279, 697, 747

sunday, 679, 769

superclass\_delegating\_accessors, 671

supports\_count\_distinct?, 170

supports\_migrations?, 170

suppress, 714

swapcase, 727

Symbol

    extensions, 759

    usage versus strings, 141

## T

t, 401

table\_name\_prefix=, 133

table\_name\_suffix=, 133

tableize, 755

tables, 171

tag, 394–395

tagged, 759

telephone\_field, 370

telephone\_field\_tag, 384

template\_engine, 11

Templates. See View templates

terabyte / terabytes, 731

test\_framework, 11

text\_area, 370–371

text\_area\_tag, 384

text\_field, 371

text\_field\_tag, 384

third, 652

thread\_variable?, 763

thread\_variable\_get, 763

thread\_variable\_set, 763

thread\_variables, 763

tidy\_bytes, 727–728

Time

    extensions, 763–773

    input tags, 350–351

    reporting distances in time, 355–356

    storing in database, 185

Time Zones

    DateTime conversions, 681–682

    default, 10

    option tags helper, 375–379

    TimeZone class, 774–778

    TimeWithZone class, 773–774

time

    time\_field, 371

    time\_field\_tag, 384

    time\_select, 351

    time\_tag, 356

    time\_zone\_options\_for\_select, 373,  
        378–379

    time\_zone\_select, 373

timestamps, 181

titleize, 755

to, 652, 720

    to\_a, 634

    to\_constant\_name, 691

    to\_date, 367, 685, 750

## to (continued)

- to\_datetime, 685, 750
- to\_default\_s, 653
- to\_f, 685
- to\_formatted\_s, 652–653, 659, 681, 686, 731–732, 746, 770–771
- to\_h, 758
- to\_hash, 634
- to\_i, 686
- to\_json, 87, 107, 553, 741
- to\_key, 629
- to\_model, 629
- to\_options, 702
- to\_param, 629, 657, 704, 741
- to\_partial\_path, 629
- to\_query, 704, 741
- to\_s, 653, 659, 771, 777
- to\_sentence, 653
- to\_sql, 265–266
- to\_time, 681, 750
- to\_xml method, 611–620
- to\_yaml, 779
- today, 679, 769, 777
- toggle, 271
- toggle!, 271
- tomorrow, 680, 769
- touch, 210
- transform\_keys, 703
- translate, 401, 407–408, 410–412
- transliterate, 710–711
- travel, 762
- travel\_to, 762
- truncate, 398–399, 751
- Truncating text, 398–399
- try, 741–742
- Turbolinks, 551–553

## U

- ugly, 448
- uncountable, 709–710
- undefine\_attribute\_method, 627
- underscore, 755
- unhook!, 691
- Unicode
  - methods for handling, 727–729
  - multibyte safe proxy for, 756
  - pluralization rules, 412–413
  - security concerns of, 392
- uniq, 164, 204
- unloadable, 743
- Unobtrusive JavaScript (UJS), 547–550
  - helpers, 548
  - jQuery UJS custom events, 550
- unscope, 164–165
- until, 696, 737
- update, 169, 271
- update\_all, 147, 188, 271
- update\_attribute, 148–149, 260–261
- update\_column(s), 260–261, 271
- update\_counters, 271
- updated\_at, 186
- updated\_on, 186
- URL
  - generation, 43
  - patterns in routing, 42
  - segment keys, 43–44
- url\_field, 371
- url\_field\_tag, 384–385
- url\_for, 113, 358
- us\_utf8?, 757
- us\_zones, 379, 778
- use\_zone, 772
- usec, 686
- utc, 684

utc?, 684  
utc\_offset, 684, 778  
utc\_to\_local, 778  
utf8\_enforcer\_tag, 384–385

## V

valid?, 146, 241–242, 256, 635, 643  
validate, 210–211, 218, 260, 649  
validate!, 648  
validates, 647–648  
    validates\_absence\_of, 242, 643  
    validates\_acceptance\_of, 242–243, 643  
    validates\_associated, 243–244  
    validates\_callbacks, 643–644  
    validates\_confirmation\_of, 244, 644  
    validates\_each, 244–245, 643  
    validates\_exclusion\_of, 246, 644  
    validates\_format\_of, 245–246, 645–646  
    validates\_inclusion\_of, 246, 645  
    validates\_length\_of, 246–247, 645–646  
    validates\_numericality\_of, 247–248, 646–647  
    validates\_presence\_of, 248–249, 647  
    validates\_uniqueness\_of, 249–251  
    validates\_with, 251–252, 648  
Validation. See Active Record, validations  
validators, 648  
validators\_on, 648  
Value objects, 299–302  
View templates  
    capturing block content, 395  
    concat method, 395s  
    cycling content, 398  
    encapsulating logic in helper

    methods, 423–424  
    highlighting content, 396  
    localization, 399  
    transforming text into HTML, 398  
    translation. See Internationalization.  
    word wrap, 399  
    See also Action View, 313–329

values, 635  
verify, 489, 716  
verify!, 171  
video\_path, 342  
video\_tag, 342–343

## W

warn, 30, 693  
warnings\_on\_first\_load, 688  
Web 2.0, 355, 431, 558  
Web architecture, 63–64  
week / weeks, 738  
    week\_field, 371  
    week\_field\_tag, 385  
    weeks\_ago, 680, 769  
    weeks\_since, 680, 769  
where, 211–212, 218–219, 236  
will\_unload?, 691  
with\_indifferent\_access, 701, 705  
with\_options, 255, 742  
word\_wrap, 399  
write, 661, 665  
write\_attribute, 135, 140

## X

XML, 611–625  
    Active Record associations, 614–617  
    customizing output, 613–614  
    extra elements, 618  
    overriding, 620

## XML (continued)

parsing, 622–624

Ruby hashes, 616

to\_xml method, 611–620

typecasting, 624

XML Builder, 620–622

XMLHttpRequestObject, 545–546

XMLMini module, 778–780

x\_sendfile\_header=, 24, 573

**Y**

y, 101

YAML, 26, 136

Devise, 460

Resque, 537–538

Sidekiq, 535

translations, 402, 408, 413

year / years, 738

years\_ago, 680, 769

years\_since, 680, 769

yesterday, 680, 770

yield, 119–120, 315–316

**Z**

zone, 772

zone=, 772–773

*This page intentionally left blank*

# REGISTER



## THIS PRODUCT

[informit.com/register](http://informit.com/register)

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **[informit.com/register](http://informit.com/register)** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

### About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

**informIT.com**

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram  
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

PEARSON

**InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

✦ Addison-Wesley

Cisco Press

EXAM<sup>4</sup>CRAM

IBM  
Press

que

PRENTICE  
HALL

SAMS

Safari<sup>®</sup>

## LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit **[informit.com/newsletters](http://informit.com/newsletters)**.
- Access FREE podcasts from experts at **[informit.com/podcasts](http://informit.com/podcasts)**.
- Read the latest author articles and sample chapters at **[informit.com/articles](http://informit.com/articles)**.
- Access thousands of books and videos in the Safari Books Online digital library at **[safari.informit.com](http://safari.informit.com)**.
- Get tips from expert blogs at **[informit.com/blogs](http://informit.com/blogs)**.

Visit **[informit.com/learn](http://informit.com/learn)** to discover all the ways you can access the hottest technology content.

### Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **[informit.com/socialconnect](http://informit.com/socialconnect)**.

