

Addison-Wesley Professional Ruby Series



Fully
up-to-date for
Rails 3

RAILS™ ANTIPATTERNS

BEST PRACTICE RUBY ON
RAILS™ REFACTORING

CHAD PYTEL ■ TAMMER SALEH

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.

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

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

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress cataloging-in-publication data is on file with the Library of Congress

Copyright © 2011 Pearson Education, Inc.

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

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

ISBN-13: 978-0-321-60481-1

ISBN-10: 0-321-60481-4

Text printed in the United States on recycled paper at RR Donnelley
in Crawfordsville, Indiana.

First printing, November 2010

Editor-in-Chief
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Copy Editor
Kitty Wilson

Indexer
The CIP Group

Proofreader
Linda Begley

Technical Reviewers
Jennifer Lindner
Pat Allen
Joe Ferris
Stephen Caudill
Tim Pope
Robert Pitts
Jim "Big Tiger" Remsik
Lar Van Der Jagt

Publishing Coordinator
Kim Boedigheimer

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

Contents

Foreword xi

Introduction xiii

Acknowledgments xvii

About the Authors xix

1 Models 1

AntiPattern: Voyeuristic Models 2

Solution: Follow the Law of Demeter 3

Solution: Push All `find()` Calls into Finders on the Model 7

Solution: Keep Finders on Their Own Model 10

AntiPattern: Fat Models 14

Solution: Delegate Responsibility to New Classes 15

Solution: Make Use of Modules 21

Solution: Reduce the Size of Large Transaction Blocks 24

AntiPattern: Spaghetti SQL 31

Solution: Use Your Active Record Associations and Finders Effectively 32

Solution: Learn and Love the `scope` Method 36

Solution: Use a Full-Text Search Engine 42

AntiPattern: Duplicate Code Duplication 50

Solution: Extract into Modules 50

Solution: Write a Plugin 59

Solution: Make Magic Happen with Metaprogramming 64

2 Domain Modeling 73

AntiPattern: Authorization Astronaut 74

Solution: Simplify with Simple Flags 76

- AntiPattern: The Million-Model March 79
 - Solution: Denormalize into Text Fields 79
 - Solution: Make Use of Rails Serialization 82

3 Views 89

- AntiPattern: PHPitis 91
 - Solution: Learn About the View Helpers That Come with Rails 92
 - Solution: Add Useful Accessors to Your Models 98
 - Solution: Extract into Custom Helpers 100
- AntiPattern: Markup Mayhem 107
 - Solution: Make Use of the Rails Helpers 109
 - Solution: Use Haml 111

4 Controllers 117

- AntiPattern: Homemade Keys 118
 - Solution: Use Clearance 119
 - Solution: Use Authlogic 121
- AntiPattern: Fat Controller 123
 - Solution: Use Active Record Callbacks and Setters 123
 - Solution: Move to a Presenter 142
- AntiPattern: Bloated Sessions 154
 - Solution: Store References Instead of Instances 154
- AntiPattern: Monolithic Controllers 161
 - Solution: Embrace REST 161
- AntiPattern: Controller of Many Faces 167
 - Solution: Refactor Non-RESTful Actions into a Separate Controller 167
- AntiPattern: A Lost Child Controller 170
 - Solution: Make Use of Nested Resources 173
- AntiPattern: Rat's Nest Resources 180
 - Solution: Use Separate Controllers for Each Nesting 181
- AntiPattern: Evil Twin Controllers 184
 - Solution: Use Rails 3 Responders 186

5 Services 189

- AntiPattern: Fire and Forget 190
 - Solution: Know What Exceptions to Look Out For 190

- AntiPattern: Sluggish Services 195
 - Solution: Set Your Timeouts 195
 - Solution: Move the Task to the Background 195
- AntiPattern: Pitiful Page Parsing 197
 - Solution: Use a Gem 198
- AntiPattern: Successful Failure 201
 - Solution: Obey the HTTP Codes 203
- AntiPattern: Kraken Code Base 207
 - Solution: Divide into Confederated Applications 207

6 Using Third-Party Code 211

- AntiPattern: Recutting the Gem 213
 - Solution: Look for a Gem First 213
- AntiPattern: Amateur Gemologist 214
 - Solution: Follow TAM 214
- AntiPattern: Vendor Junk Drawer 216
 - Solution: Prune Irrelevant or Unused Gems 216
- AntiPattern: Miscreant Modification 217
 - Solution: Consider Vendored Code Sacrosanct 217

7 Testing 221

- AntiPattern: Fixture Blues 223
 - Solution: Make Use of Factories 225
 - Solution: Refactor into Contexts 228
- AntiPattern: Lost in Isolation 236
 - Solution: Watch Your Integration Points 238
- AntiPattern: Mock Suffocation 240
 - Solution: Tell, Don't Ask 241
- AntiPattern: Untested Rake 246
 - Solution: Extract to a Class Method 248
- AntiPattern: Unprotected Jewels 251
 - Solution: Write Normal Unit Tests Without Rails 251
 - Solution: Load Only the Parts of Rails You Need 254
 - Solution: Break Out the Atom Bomb 259

8 Scaling and Deploying 267

AntiPattern: Scaling Roadblocks 268

Solution: Build to Scale from the Start 268

AntiPattern: Disappearing Assets 271

Solution: Make Use of the System Directory 271

AntiPattern: Sluggish SQL 272

Solution: Add Indexes 272

Solution: Reassess Your Domain Model 277

AntiPattern: Painful Performance 282

Solution: Don't Do in Ruby What You Can Do in SQL 282

Solution: Move Processing into Background Jobs 286

9 Databases 291

AntiPattern: Messy Migrations 292

Solution: Never Modify the up Method on a Committed Migration 292

Solution: Never Use External Code in a Migration 293

Solution: Always Provide a down Method in Migrations 295

AntiPattern: Wet Validations 297

Solution: Eschew Constraints in the Database 298

10 Building for Failure 301

AntiPattern: Continual Catastrophe 302

Solution: Fail Fast 302

AntiPattern: Inaudible Failures 306

Solution: Never Fail Quietly 307

Index 311

Foreword

It's hard to believe that it will soon be three years since Zed Shaw published his infamous (and now retracted) rant "Rails Is a Ghetto." Even though Zed's over-the-top depiction of certain well-known people was wicked and pure social satire, the expression he coined has stuck like the proverbial thorn among certain higher echelons of the community. It's an especially piquant expression to use when we're called on to fix atrocious Rails projects. Occasionally, we'll even use the phrase with regard to our own messes. But most commonly, this expression is applied to code written by the *unwashed masses*. The rapid ascension of Rails as a mainstream technology has attracted droves of eager programmers from both outside and inside the wide sphere of web development. Unfortunately, Rails doesn't discriminate among newcomers. It offers deep pitfalls for bearded wise men of the object-oriented world and PHP script kiddies alike.

Frankly, I would have written this book myself eventually, because there's such a need for it in the marketplace. At Hashrocket, we do a lot of project rescue work. Oh, the agony! We've seen every AntiPattern detailed in this book rear its ugly face in real-life projects. Sometimes we see almost every AntiPattern in this book in a *single* project! My good friends and consultants extraordinaire Chad and Tammer have seen the same horrors. Only fellow consultants like these two could write this book properly because of the wide variety of coding challenges we face regularly. The solutions in this book cover a wide range of sticky situations that we know any professional Ruby developer will run into on a regular basis.

If you're new to Rails (and, based on the demographics, you probably are), then you're now holding one of the most valuable resources possible for getting past the chasm that separates an ordinary Rails developer from greatness. Congratulations and good luck making the leap.

—Obie Fernandez

Author of *The Rails 3 Way*

Series editor of the Addison-Wesley Professional Ruby Series

CEO and founder of Hashrocket

This page intentionally left blank

Introduction

As Rails consultants, we've seen a lot of Rails applications. The majority of the AntiPatterns described in this book are directly extracted from real-world applications. We hope that by formalizing their descriptions here, we can present you with the tools you'll need to identify these AntiPatterns in your own code, understand their causes, and be able to refactor yourself out of the broken patterns.

What Are AntiPatterns?

AntiPatterns are common approaches to recurring problems that ultimately prove to be ineffective.

The term *AntiPatterns* was coined in 1995 by Andrew Koenig, inspired by Gang of Four's book *Design Patterns*, which developed the concept of design patterns in the software field. The term was widely popularized three years later by the book *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (William Brown, Raphael Malveau, Skip McCormick, and Tom Mowbray). According to the authors of *AntiPatterns*, there must be at least two key elements present to formally distinguish an actual AntiPattern from a simple bad habit, bad practice, or bad idea:

- A repeated pattern of action, process, or structure that initially appears to be beneficial but ultimately produces more bad consequences than beneficial results
- A refactored solution that is clearly documented, proven in actual practice, and repeatable

What Is Refactoring?

Refactoring is the act of modifying an application's code not to change its functional behavior but instead to improve the quality of the application itself. These improvements

are intended to improve readability, reduce complexity, increase maintainability, and improve the extensibility (that is, possibility for future growth) of the system.

This book makes extensive reference to the process of refactoring in order to fix code that is exhibiting an AntiPattern. In an attempt to increase readability and understandability of the AntiPatterns and solutions in this book, we've left out the automated test suite that should accompany the code. We want to draw extra attention to the fact that your code should be well tested. When you have tests in place, some of the solutions we've presented will be much easier to implement with confidence. Without tests, some of the solutions might not even be possible. Unfortunately, many of the applications you encounter that exhibit these AntiPatterns will also be untested.

How to Read This Book

Each AntiPattern in this book outlines the mistakes we see in the wild and the negative effects they have on developer velocity, code clarity, maintenance, and other aspects of a successful Rails project. We follow each AntiPattern with one or more solutions that we have used in practice and that have been proven as proper fixes for the AntiPattern.

While you can read this book straight through from front to back, we've taken great pains to make each solution stand on its own. Therefore, this book is both a strong technical publication as well as a quick source of reference for Rails developers looking to hone their techniques in the trenches.

The following is a brief outline of what's covered in each chapter:

- **Chapter 1, “Models”:** Because Rails encourages code to be pushed down the Model-View-Controller (MVC) stack to the Model layer, it's fitting that a chapter on models is the largest chapter in the book. Here, we focus on a variety of AntiPatterns that occur in Model layer code, from general object-oriented programming violations to complex SQL and excessive code duplication.
- **Chapter 2, “Domain Modeling”:** Going beyond the nitty-gritty code at the Model layer in a Rails project, this chapter focuses on overall schema and database issues. This chapter covers issues such as normalization and serialization.
- **Chapter 3, “Views”:** The Rails framework gives developers a large number of tools and conventions that make code in the Model and Controller layers consistent and maintainable. Unfortunately, the required flexibility in the View layer

prevents this sort of consistency. This chapter shows how to make use of the View layer tools Rails provides.

- **Chapter 4, “Controllers”:** Since the integration of a RESTful paradigm in the Rails framework, the Controller layer has seen some significant improvements. This chapter goes through the AntiPatterns we’ve seen in Controller-layer-related Rails code.
- **Chapter 5, “Services”:** Dealing with and exposing APIs requires tenacity. This chapter walks through all the common pitfalls we’ve seen, including timeouts, exceptions, backgrounding, response codes, and more.
- **Chapter 6, “Using Third-Party Code”:** This short chapter reviews some of the AntiPatterns that can come from incorporating community plugins and gems into your applications.
- **Chapter 7, “Testing”:** One of the strengths of Rails is the strong push toward test-driven development. Unfortunately, we’ve seen as many AntiPatterns inside test suites as in production code. This chapter outlines these AntiPatterns and how to address them.
- **Chapter 8, “Scaling and Deploying”:** Developing a Rails application locally is a great experience, but there are many factors to consider once it’s time to release an application to the world. This chapter will help you ensure that your applications are ready for prime time.
- **Chapter 9, “Databases”:** This chapter outlines the common issues we’ve seen with migrations and validations.
- **Chapter 10, “Building for Failure”:** Finally, the last chapter in the book gives guidance on general best practices for ensuring that an application degrades gracefully once it encounters the real world.

This page intentionally left blank

CHAPTER 9

Databases

With the Rails framework providing a simple ORM that abstracts many of the database details away from the developer, the database is an afterthought for many Rails developers. While the power of the framework has made this okay to a certain extent, there are important database and Rails-specific considerations that you shouldn't overlook.

AntiPattern: Messy Migrations

Ruby on Rails database migrations were an innovative solution to a real problem faced by developers: How to script changes to the database so that they could be reliably replicated by the rest of the team on their development machines as well as deployed to the production servers at the appropriate time. Before Rails and its baked-in solution, developers often wrote ad hoc database change scripts by hand, if they used them at all.

However, as with most other improvements, database migrations are not without pain points. Over time, a database migration can become a tangle of code that can be intimidating to work with rather than the joy it should be. By strictly keeping in mind the following solutions, you can overcome these obstacles and ensure that your migrations never become irreconcilably messy.

Solution: Never Modify the `up` Method on a Committed Migration

Database migrations enable you to reliably distribute database changes to other members of your team and to ensure that the proper changes are made on your server during deployment.

If you commit a new migration to your source code repository, unless there are irreversible bugs in the migration itself, you should follow the practice of never modifying that migration. A migration that has already been run on another team member's computer or the server will never automatically be run again. In order to run it again, a developer must go through an orchestrated dance of backing the migration down and then up again. It gets even worse if other migrations have since been committed, as that could potentially cause data loss.

Yes, if you're certain that a migration hasn't been run on the server, then it's possible to communicate to the rest of the team that you've changed a migration and have them re-migrate their database or make the required changes manually. However, that's not an effective use of their time, it creates headaches, and it's error prone. It's simply best to avoid the situation altogether and never modify the `up` method of a migration.

Of course, there will be times when you've accidentally committed a migration that has an irreversible bug in it that must be fixed. In such circumstances, you'll have no choice but to modify the migration to fix the bug. Ideally, the times when this happen are few and far between. In order to reduce the chances of this happening, you

should always be sure to run the migration and inspect the results to ensure accuracy *before* committing the migration to your source code repository. However, you shouldn't limit yourself to simply running the migration. Instead, you should run the migration and then run the down of the migration and rerun the up. Rails provides rake tasks for doing this:

```
rake db:migrate
rake db:migrate:redo
```

The rake `db:migrate:redo` command runs the `down` method on the last migration and then reruns the `up` method on that migration. This ensures that the entire migration runs in both directions and is repeatable, without error. Once you've run this and double-checked the results, you can commit your new migration to the repository with confidence.

Solution: Never Use External Code in a Migration

Database migrations are used to manage database change. When the structure of a database changes, very often the data in the database needs to change as well. When this happens, it's fairly common to want to use models inside the migration itself, as in the following example:

```
class AddJobsCountToUser < ActiveRecord::Migration
  def self.up
    add_column :users, :jobs_count, :integer, :default => 0
    Users.all.each do |user|
      user.jobs_count = user.jobs.size
      user.save
    end
  end

  def self.down
    remove_column :users, :jobs_count
  end
end
```

In this migration above, you're adding a counter cache column to the `users` table, and this column will store the number of jobs each user has posted. In this migration,

you're actually using the `User` model to find all users and update the column of each one. There are two problems with this approach.

First, this approach performs horribly. The code above loads all the users into memory and then for each user, one at a time, it finds out how many jobs each has and updates its count column.

Second, and more importantly, this migration does not run if the model is ever removed from the application, becomes unavailable, or changes in some way that makes the code in this migration no longer valid. The code in migrations is supposed to be able to be run to manage change in the database, in sequence, at any time. When external code is used in a migration, it ties the migration code to code that is not bound by these same rules and can result in an unrunnable migration.

Therefore, it's always best to use straight SQL whenever possible in your migrations. If you do so, you can rewrite the preceding migration as follows:

```
class AddJobsCountToUser < ActiveRecord::Migration
  def self.up
    add_column :users, :jobs_count, :integer, :default => 0
    update(<<-SQL)
      UPDATE users SET jobs_count = (
        SELECT count(*) FROM jobs
        WHERE jobs.user_id = users.id
      )
    SQL
  end

  def self.down
    remove_column :users, :jobs_count
  end
end
```

When this migration is rewritten using SQL directly, it has no external dependencies beyond the exact state of the database at the time the migration should be executed.

There may be cases in which you actually do need to use a model or other Ruby code in a migration. In such cases, the goal is to rely on no external code in your migration. Therefore, all code that's needed, including the model, should be defined inside the migration itself. For example, if you really want to use the `User` model in the preceding migration, you rewrite it like the following:


```
class AddJobsCountToUser < ActiveRecord::Migration
  class Job < ActiveRecord::Base
  end
  class User < ActiveRecord::Base
    has_many :jobs
  end

  def self.up
    add_column :users, :jobs_count, :integer, :default => 0
    User.reset_column_information
    Users.all.each do |user|
      user.jobs_count = user.jobs.size
      user.save
    end
  end

  def self.down
    remove_column :users, :jobs_count
  end
end
```

Since this migration defines both the `Job` and `User` models, it no longer depends on an external definition of those models being in place. It also defines the `has_many` relationship between them and therefore defines everything it needs to run successfully. In addition, note the call to `User.reset_column_information` in the `self.up` method. When models are defined, Active Record reads the current database schema. If your migration changes that schema, calling the `reset_column_information` method causes Active Record to re-inspect the columns in the database.

You can use this same technique if you must calculate the value of a column by using an algorithm defined in your application. You cannot rely on the definition of that algorithm to be the same or even be present when the migration is run. Therefore, the algorithm should be duplicated inside the migration itself.

Solution: Always Provide a down Method in Migrations

It's very important that a migration have a reliable `self.down` defined that actually reverses the migration. You never know when something is going to be rolled back. It's truly bad practice to not have this defined or to have it defined incorrectly.

Some migrations simply cannot be fully reversed. This is most often the case for migrations that change data in a destructive manner. If this is the case for a migration for which you're writing the down method, you should do the best reversal you can do. If you are in a situation where there is a migration that under no circumstances can ever be reversed safely, you should raise an `ActiveRecord::IrreversibleMigration` exception, as shown here:

```
def self.down
  raise ActiveRecord::IrreversibleMigration
end
```

Raising this exception causes migrations to be stopped when this down method is run. This ensures that the developer running the migrations understands that there is something irreversible that has been done and that cannot be undone without manual intervention.

Once you have the down method defined, you should run the migration in both directions to ensure proper functionality. As discussed earlier in this chapter, in the section “Solution: Never Modify the up Method on a Committed Migration,” Rails provides rake tasks for doing this:

```
rake db:migrate
rake db:migrate:redo
```

The rake `db:migrate:redo` command runs the down method on the last migration and then reruns the up method on that migration.

AntiPattern: Wet Validations

Ruby on Rails generally treats a database as a dumb storage device, essentially working only with many of the common-denominator features found in all the databases it supports and eschewing additional database functionality such as foreign keys and constraints. But many Rails developers eventually realize that a database has this functionality built in, and they attempt to use it by trying to duplicate the validation and constraints from their models into the database. For example, the following `User` model has a number of validations:

```
class User < ActiveRecord::Base
  validates :account_id, :presence => true
  validates :first_name, :presence => true
  validates :last_name, :presence => true

  validates :password, :presence => true,
                  :confirmation => true,
                  :if => :password_required?

  validates :email, :uniqueness => true,
                :format => { :with => %r{.+@.+\.+} },
                :presence => true

  belongs_to :account
end
```

You could attempt to create a database table to back this model that attempts to enforce the same validations at the database level, using database constraints. The (inadequate) migration to create that table might look something like this:

```
self.up
  create_table :users do |t|
    t.column :email, :string, :null => false
    t.column :first_name, :string, :null => false
    t.column :last_name, :string, :null => false
    t.column :password, :string
    t.column :account_id, :integer
  end
  execute "ALTER TABLE users ADD UNIQUE (email)"
```

```
execute "ALTER TABLE users ADD CONSTRAINT
user_constrained_by_account FOREIGN KEY (account_id) REFERENCES
accounts (id) ON DELETE CASCADE"
end

self.down
execute "ALTER TABLE users DROP FOREIGN KEY
user_constrained_by_account"
drop_table :users
end
```

However, there are several reasons this doesn't work in practice. For one thing, not all databases support all the constraints that Active Record supports. For example, in MySQL, it's possible to enforce the uniqueness constraints on email, but none of the other constraints are fully possible without the use of stored procedures and triggers. For example, in the migration earlier in this chapter, there is only a constraint on NULL values in the `first_name` column. A blank string would still be allowed to be inserted.

If you are on a database that supports these constraints, you are then left to maintain them all by hand, in duplicate—a process that is tedious and error prone.

Active Record does not handle violations of database constraints well. It does not automatically read the constraints in the database. And if something is out of sync and a constraint in the database is hit, this will result in an exception that is not handled gracefully at the library level. The result is a failure the user sees or one that the programmer must handle, which is impractical.

Solution: Eschew Constraints in the Database

It's simply best to not fight the opinion of Active Record that database constraints are declared in the model and that the database should simply be used as a datastore.

Despite all of the above, you may find yourself working with a DBA who insists that foreign key constraints or other constraints be stored in the database, or you yourself may simply believe in this principle. In such a case, it is strongly recommended that you not attempt to do this by hand and instead use a plugin that provides support for this. One such plugin is *Foreigner* (<http://github.com/matthuhiggins/foreigner/>), which provides support for managing foreign key constraints in migrations. Several other well-supported plugins provide support for additional constraints, most of which will be specific to your database server.

There's Always an Exception

In the example we've been looking at in this section, the exception is `NULL` constraints coupled with default database values. Active Record handles these constraints perfectly, with the defaults even being picked up and populated in your model automatically. Therefore, the recommended way to provide default values to your model attributes is by storing the default values in the database. For example, if you want to default a Boolean column to `true`, you can do so in the database:

```
add_column :users, :active, :boolean, :null => false, :default =>
true
```

This will result in the `active` attribute on the user model being set to `true` whenever a new user is created:

```
>> user = User.new
>> user.active?
=> true
```

You can use this swell behavior to your benefit to simplify code and make your objects more consistent. In most applications, setting all Booleans to allow `null` and to default to `false` is preferred. That way, your Booleans will really have only two possible values, `true` and `false`, not `true`, `false`, and `nil`.

Index

=, 114
character, 114
% character, 114
[] operator, 115
. character, 114
- operator, 114

A

Abstract methods, 57
Accessors, 98–100
AccountsController, 26, 29, 144, 148
AccountsControllerTest, 143, 146
ActionMailer, 193, 254
ActionPack, 254
ActivationsController, 164
Active Presenter, 149–153
Active Record
 associations, 3–7, 11–12, 32–36, 242, 282
 lifecycle methods, 125
 scope, 33, 36–42
 scopes, 10–13
 validation macros, 53
ActiveRecord#save!, 226
ActiveResource, 254
ActiveSupport, 255
 ActiveSupport::Concern, 53, 56, 58
 ActiveSupport::TestCase, 223, 226,
 236, 246, 248, 255, 261
 Acts As Revisionable, 85
 ActsAsVersioned, 85
 add_user, 75–76
 admin, 27–29
 AdminController, 161–165
 AlertsController, 100
 alerts_rss_url, 104–105
 Amateur Gemologist, 214–215
 Amazon S3, 268–270
 AND, 44
 Antipatterns, xiii
 *AntiPatterns: Refactoring Software,
 Architectures, and Projects in Crisis*, xiii
 APIs, 184–186
 /app/helpers, 91
 ApplicationHelper, 103
 application_helper.rb, 103
 /app/views directory, 91
 apt, 44
 @article.comments.count, 282
 @article.comments.length, 282
 @article.comments.size, 282
 ArticlesController, 123–125,
 139–142, 243–245
 assert, 221, 233

- Association methods, 33–36
- AssociationProxy, 11–13, 141
- Associations, 3–7, 11–12, 242, 282
- Attributes, 27–28, 47–48
- `:authenticate` before filter, 119–120
- Authentications, 118–122, 167–169
- Authlogic, 121–123
- Authorization, 118
- Authorization Astronaut, 74–78
- Automated test suite, xiv

B

- Background processing, 286–289
- Background tasks, 195–196
- Backtrace, 86–87
- Beck, Kent, 221
- `before_create`, 28–29, 86
- `before_filter`, 173, 304
- `before_save`, 150, 309
- Behavior-driven development (BDD), 221, 225–233
- Blawg, 257–264
- Bloated Sessions, 154–160
- Boolean columns, 276
- Boolean values, 299
- Booleans, 76–78, 84–85
- Bundler, 63, 219

C

- Callbacks, 25–27, 237, 309
- Callbacks and setters, 123–142
- `can_` methods, 74–76
- Capistrano, 271
- Case-insensitive sort, 283
- Catlin, Hampton, 112
- `city` column, 275
- Class (defined), 2

- Class, compared with model, 2
- `class` attributes, 96, 109, 114–115
- Class definitions, 34, 64, 68, 70, 225
- Class method rake tests, 248–250
- Classes, refactoring, 16–17
- Clearance, 119–121
- Cloud deployment, 268–270
- Clustered environment, 270
- Code patching, 217–219
- Code sharing, 219
- Code splitting, 207–210
- Comments, 273–274
- Complexity, 14
- `composed_of` method, 19–20
- Composite index, 274–276
- Composition, 18–19, 61
- Conditional callbacks, 137–138
- Conditional joins, 273–274
- Conditions, 40, 44
- `config.plugin_paths`, 261
- `config/routes.rb`, 120, 168, 264
- Confirmation email, 28–29
- Constants, 57, 243
- Constraints, 297–299
- `content_for`, 95–98, 115
- `content_tag`, 102, 110–111
- `content_tag_for`, 110
- Contexts, 230–232, 234–235
- Continual Catastrophe, 302–305
- Controller actions, 161
- Controller naming, 162
- Controllers
 - Bloated Sessions, 154–160
 - Controller of Many Faces, 167–169
 - Evil Twin Controllers, 184–188
 - Fat Controller, 123–153
 - Homemade Keys, 118–122
 - Lost Child Controller, 170–179
 - Monolithic Controllers, 161–166
 - Rat's Nest Resources, 180–183
- Conversion, 16–17
- Cookies, 154–156, 198

count, 282
 create, 124–130, 141
 #create, 143–144, 146–148
 create_account!, 24–29
 created_at, 47–48, 128
 create_first_version!, 126–133
 create_version!, 124, 126–128,
 130–133, 137–139
 cron, 286–287, 302
 CSS, 91, 107–109, 115–116
 CSS3, 198–199
 csv, 17
 Cucumber, 120–121, 163, 239
 current_version, 133–138

D

Database

Boolean values, 299
 constraints, 297–299
 defaults, 129, 147, 277
 down method, 293–296
 external code, 293–295
 indexes, 272–277
 Messy Migrations, 292–296
 transactions, 125, 147
 up method, 292–296
 validations, 297–299
 Wet Validations, 297–299
 database.yml, 256, 260
 DateTime columns, 276
 db:indexes:missing, 277
 db:migrate, 293, 296
 db:migrate:redo, 293, 296
 define_method, 68, 77, 238
 delayed_job (DJ), 196, 287–289
 delegate method, 6–7, 18–19
 delete_user, 76, 162–163, 166
 Delta indexes, 48–49
 Denormalization, 79–82, 84, 280–281
 Denormalized role_type, 77

Deploying. *See* Scaling and deploying
 Descriptive naming, 134
Design Patterns: Elements of Reusable Object-Oriented Software, xiii, 54, 225
 Disappearing Assets, 271
 <div>, 114
 div_for, 110, 115
 Domain model, 277–281, 284
 Domain modeling
 Authorization Astronaut, 74–78
 Million-Model March, 79–87
 dom_class, 110
 dom_id, 110
 down method, 293–296
 DRY (Don't Repeat Yourself) Principle,
 50, 226
 Duplicate Code Duplication
 metaprogramming, 64–71
 modules, 51–59
 plugins, 59–64
 Duplicate exceptions, 85
 Dust, 221

E

Eager loading, 280
 Email confirmation, 28–29
 Email errors, 192–193
 Email model, 208
 EmailsController, 208–210
 Encapsulation, 2, 4–7
 Engine Yard, 270
 Engines, 119
 Environment info, 86–87
 environment.rb, 261
 ERb, 89, 91, 94–95
 Error catching application, 85–86
 Error logging, 192–194, 309–310
 Error pages, 301, 308
 Errors, rescuing, 190–192, 308
 eval, 63

- Evaluating third-party tools, 214–215
- Evil Twin Controllers, 184–188
- Exceptional, 194, 309–310
- exception_notification, 194, 310
- Exceptions, 85, 125, 148, 190–194, 309
- EXPLAIN statements, 277
- extend, 22–23, 34–35, 50, 53, 56, 58
- External code, 293–295
- :extract_backtrace_info, 86–87
- :extract_environment_info, 86–87
- :extract_request_info, 86–87
- Extracting code into modules, 50–59

F

- Facebook errors, 190–192
- Factories, 225–227, 234–235
- Factory, 225
- Factory.define, 227
- FactoryGirl gem, 227, 234, 256
- Factory.next, 227
- Factory.sequence, 227
- Fail fast, 302–305
- Fail whale, 301
- Failure
 - Continual Catastrophe, 302–305
 - Inaudible Failures, 306–310
- FakeWeb, 250
- Fat Controller
 - callbacks and setters, 123–142
 - conditional callbacks, 137–138
 - create, 124–130
 - current_version, 133–138
 - database transactions, 125
 - exceptions, 125
 - lifecycle methods, 125
 - presenter, 142–153
 - unless, 135
- Fat Models
 - composed_of method, 19–20
 - delegate method, 18–19

- large transaction blocks, 24–30
- Law of Demeter, 18
- modules, 21–24
- nested attributes, 27–28
- refactor into new classes, 15–21
- Single Responsibility Principle, 15–21
- Ferret, 43
- Fielding, Roy, 161
- Fields, Jay, 149
- fields_for, 84, 145
- File attachment plugin, 268–270
- Filesystem limits, 269
- FileUtils::NoWrite, 250
- Filters, 47
- find() calls, 7–13
- find_by_sql, 34*n*
- Fire and Forget
 - exceptions, 190–194
 - Hoptoad, 192–194
 - HTTP errors, 192
 - publish_to, 190–191
- 500 error, 190, 303, 308
- Fixture Blues
 - contexts, 228–235
 - factories, 225–227
- fixtures, 223
- flatten, 284–285
- Flow control, 125
- Foreign keys, 273, 277, 297–298
- Foreigner, 298
- Forking, 247
- Forking gems, 219
- :format, 100–101
- Formatted URL helpers, 100–101
- form_for, 92–93, 110, 145, 175
- Full-text search engine, 42–49

G

- Gamma, Erich, 54, 225
- Gang of Four, xiii

Gem install, 44
 Gems
 Authlogic, 121–123
 Clearance, 119–121
 compared with plugins, 63–64, 211
 evaluating, 214–215
 forking, 219
 git repository, 219
 Haml, 112–114
 modifying, 217
 monkey patching, 217–219
 parsing web pages, 198–200
 unused, 216
 when to look for, 213
 generate plugin, 60–61
 Generators, 60–61, 119–120
 Get, 75–76, 244, 263
 git repository, 219
 GitHub, 211–219, 227, 287
 Golick, James, 149
 Google App Engine, 270
 Graceful degradation, 210

H

Haml, 111–116
 .haml, 113
 has_and_belongs_to_many, 74–75, 83
 has_finder, 40*n*
 hashed_password, 225
 has_many, 12, 77, 83, 295
 Helm, Richard, 54, 225
 Helpers, 92–98, 100–106
 Homemade Keys
 Authlogic, 121–123
 Clearance, 119–121
 Hoptoad, 85–87, 192–194, 309–310
 HTML
 Haml, 111–116
 parser, 198–199
 semantic markup, 107–109

HTTP errors, 192
 HTTP Post, 208–210
 HTTP status codes, 203–206
 Hunt, Andy, 50

I

id attributes, 109–111, 114
 id column, 272–273
 :id_partition, 269–271
 Inactive code, 214
 Inaudible Failures, 306–310
 include, 22–23, 50
 included, 53, 56, 58
 includes, 278, 280
 Indentation, 113–114
 Indexes, 272–277
 Indexing, 44–45
 Inheritance, 2
 _INITIALIZER, 24, 52
 init.rb, 60–63, 253–254
 Inline text, 114
 Instance methods, 23, 50, 61–62
 Integration points, 238–239
 Irreversible actions, 305
 Irreversible migration, 296

J

JavaScript, 89, 91, 94, 109, 111, 178
 Johnson, Ralph, 54, 225
 Joins, 273–274
 Json, 17, 184

K

Koenig, Andrew, xiii
 Kraken Code Base, 207–210

L

- lambda, 37–39
- Law of Demeter, 3–7, 18, 38
- layouts directory, 91
- length, 282
- /lib directory, 52–53, 61, 63
- Lifecycle methods, 125
- Lighthouse, 208
- Limerick Rake, 277
- link_to, 94–98, 100–102, 172
- log-queries-not-using-indexes, 277
- Lorem Ipsum, 251–254
- Lost Child Controller, 170–179
- Lost in Isolation, 236–239

M

- Macros, 53, 69, 150
- Markup helpers, 102–103
- Markup Mayhem
 - Haml, 111–116
 - Rails helpers, 109–111
 - semantic markup, 107–109
- Martin, Robert Cecil, 16, 19
- Mechanize, 198
- MessagesController, 180–183
- Messy Migrations, 292–296
- Metaprogramming, 64–71
- Method (defined), 2
- Method names, 238
- Methods, 33–36
- Migration, 129–130, 262, 292–296
- Million-Model March
 - denormalizing data, 79–82
 - serialization, 82–87
- MIME, 89
- Miscreant Modification, 217–219
- Missing indexes, 277
- Mock Suffocation, 240–245

- Mocking, 2, 39, 236–239
- Mocking and stubbing, 247
- Model, 2
- Models
 - Duplicate Code Duplication, 50–71
 - Fat Models, 14–30
 - Spaghetti SQL, 31–49
 - Voyeuristic Models, 2–13
- Model-View-Controller (MVC)
 - architecture, 2–3
- Model-View-Presenter (MVP) pattern, 149
- Modifying gems, 217
- Modularity, 2
- Modules, 21–24, 51–59
- Monkey patching, 217–219
- Monolithic Controllers, 161–166
- Multistep wizard, 154–158
- MySQL, 43–44, 277, 298

N

- N+1, 279
- named_scope, 40*n*, 42, 242
- Naming controllers, 162
- Nested attributes, 27–28
- Nested controllers, 182–183
- Nested resources, 173–179
- Nested transactions, 146–147
- Net::HTTP library timeout, 195
- Never fail quietly, 307–310
- New Relic, 194, 277, 310
- new_version, 141–142
- nil, 308
- Nokogiri, 198–199
- Normalized domain model, 79

O

- Object-oriented programming, 2–3, 16, 18
- “One assertion per test,” 233–234

- One-to-many associations, 170–179
- Open source code, patching, 217–219
- `ordersController`, 155–159
- ORM (object-relational mapping), 1–3, 73
- OS X Leopard, 260

P

- Painful Performance
 - background processing, 286–289
 - using SQL, 282–285
- Paperclip, 268–271
- params, 104, 139–141
- Parsing web pages, 197–200
- Password, 118–122, 163
- `PasswordsController`, 164–165
- Patching code, 217–219
- PDE, 17
- perform, 196, 287–288
- Performance testing, 277, 310
- PHPitis
 - accessors, 98–100
 - helpers, 100–106
 - view helpers, 92–98
- Pitiful Page Parsing
 - Mechanize, 198–199
 - Nokogiri, 198–199
 - RestClient, 199
- Pivotal Tracker, 208
- Plugins
 - Acts As Revisionable, 85
 - ActsAsVersioned, 85
 - Blawg, 257–264
 - compared with gems, 63–64, 211
 - Foreigner, 298
 - guide, 61
 - Limerick Rake, 277
 - Lorem Ipsum, 251–254
 - New Relic, 194, 277, 310
 - Paperclip, 268–271
 - QueryReviewer, 277

- Rails Footnotes, 277
- Slugalicious, 255–258
- testing, 251–265
- user authentication, 118–122
- versioning, 217–219
- writing and sharing, 59–64
- Polymorphic conditional joins, 273–274
- Polymorphism, 2
- port, 44
- `PortfoliosController`, 304
- Post, 208–210, 261, 263
- PostgreSQL, 43–44
- `PostsController`, 169, 261, 263
- `posts.yml`, 229
- `PostTest`, 226–227, 236, 261–262
- `post_test.rb`, 236
- Pragmatic Programmer, The*, 50
- Presenter Pattern, 142–153
- Primary keys, 272–273
- Principle of Least Knowledge. *See* Law of Demeter
- private, 2
- protected, 2
- public, 2
- /public/javascripts, 91
- /public/stylesheets, 91
- publish_to, 189–191

Q

- QueryReviewer, 277
- Queue systems, 209
- Queuing, 287–289

R

- Rails (library), 255
- Rails 3, 63, 186–188
- Rails Footnotes, 277
- rake command, 262

- rake db:migrate, 293, 296
- rake db:migrate:redo, 293, 296
- Rake routes, 173
- rake tasks, 44–46, 246–250, 262–264, 293, 296
- Rat’s Nest Resources, 180–183
- RAW version state, 129
- Readability, 305
- read_timeout, 195
- Recutting the Gem, 213
- Redis, 287
- Refactoring, xiii–xiv, 3–4, 15–21, 167–169, 228–235
- References, 154–160
- Relationship collections, 134
- RemoteProcess, 36–39
- render method, 94–96
- Request info, 86–87
- require, 62–63, 104
- rescue, 124, 190–192, 307–308
- rescue_from, 308
- reset_column_information, 295
- Resource, 167–169
- Responders, 186–188
- respond_to, 184–186, 188
- respond_with, 187–188
- Resque, 196, 209, 287
- RestClient, 199
- RESTful APIs, 201–202
- RESTful controllers, 161–169
- RJS, 89, 91–92
- Routes file, 264
- RPM, 277, 310
- Rspec, 221
- rss_link, 101–102, 104–105

S

- Sass, 115–116
- save, 146–147, 241–242
- save method, 125–127, 140

- save! method, 25–26, 126, 139–140, 148, 307–310
- #save!, 148, 242
- Scaling and deploying
 - Disappearing Assets, 271
 - Painful Performance, 282–289
 - Scaling Roadblocks, 268–270
 - Sluggish SQL, 272–281
- schema.rb, 256–257
- scope, 10–13, 33, 36–42, 242
- searchable, 54
- Searching, 46–48
- Searching serialized data, 85–87
- self.down, 294–296, 298
- self.up, 293–295, 297
- Semantic markup, 107–116
- send, 238
- send_confirmation_email, 28–29
- send_later, 288
- Serialization, 82–87
- serialize, 84–87
- Services
 - Fire and Forget, 190–194
 - Kraken Code Base, 207–210
 - Pitiful Page Parsing, 197–200
 - Sluggish Services, 195–196
 - Successful Failure, 201–206
- Session store, 154–160
- set_version_number, 131
- Sharding, 268
- should, 150, 231
- Shoulda, 221–222, 228
- signup (presenter), 149–153
- Simplicity, 14
- Single Responsibility Principle, 15–21
- Single-table inheritance (STI) pattern, 275
- size, 282
- Slashes, 99
- Slow query logging, 277
- Slugalicious, 255–258
- Sluggish Services
 - background tasks, 195–196
 - delayed_job, 196

- Resque, 196
- timeouts, 195
- Sluggish SQL
 - domain model, 277–281
 - indexes, 272–277
- SMTP, 193
- Solr, 43
- `SongsController`, 172–176, 184–186, 201–203
- Sorting, 47–48, 283
- Spaghetti SQL
 - Active Record associations, 32–36
 - full-text search engine, 42–49
 - Law of Demeter, 38
 - scope method, 36–42
- Sphinx, 43–44
- SQL, 37–38, 282–285
- sqlite3, 256, 260
- “SRP: The Single Responsibility Principle,” 16, 169
- Standard controller actions, 161
- `StandardError`, 192
- Star syntax, 45–46
- `state` column, 275
- `state` model, 79–82
- Stateless, 154
- Status codes, 203–206
- Stubs, 240–245
- Submodules, 61–62
- Successful Failure
 - HTTP status codes, 203–206
 - RESTful APIs, 201–202
- Superclass, 54, 57, 59
- suspenders, 24
- Symlink, 260–261, 271
- System directory, 271

T

- Tags, 107, 273–274
- Taligent, 149
- TAM (tests, activity, and maturity), 214–215
- Template pattern, 54, 56–59
- Test-driven development (TDD), 64, 221, 241, 251
- `test/factories.rb`, 256
- `test/factory.rb`, 225
- `/test/fixtures`, 170–171, 223
- `test_helper.rb`, 104, 147, 225, 252–256
- Testing
 - contexts, 230–232
 - Cucumber, 163, 239
 - embedding a Rails app, 259–265
 - Fixture Blues, 223–235
 - and integration points, 238–239
 - Lost in Isolation, 236–239
 - Mock Suffocation, 240–245
 - “one assertion per test,” 233–234
 - performance, 277, 310
 - plugins and gems, 251–265
 - rake tasks, 246–250
 - `schema.rb`, 256–257
 - stubs, 240–245
 - test cases, 221, 229–230, 240–241
 - and third-party tools, 214–215
 - Unprotected Jewels, 251–265
 - Untested Rake, 246–250
 - view helpers, 103–106
- `Test::Spec`, 221
- `Test::Unit`, 221
- `test/unit/helpers`, 104
- TextMate, 261
- Thinking Sphinx
 - delta indexes, 48–49
 - filters, 47
 - gem install, 44
 - indexing, 44–45
 - searching, 46–48
 - sorting, 47–48
 - star syntax, 45–46
- Third-party code
 - Amateur Gemologist, 214–215
 - Miscreant Modification, 217–219

Third-party code (*continued*)

- Recutting the Gem, 213
- Vendor Junk Drawer, 216
- Thomas, Dave, 50
- thoughtbot
 - Clearance, 119–121
 - FactoryGirl gem, 227, 234, 256
 - Hoptoad, 85–87, 192–194, 309–310
 - Limerick Rake, 277
 - Paperclip, 268–271
 - Shoulda, 221–222, 228
 - suspenders, 24
- Ticket, 208
- TicketsController, 208–210
- Ticket-tracking application, 208–210
- Time, 126–128
- Timeouts, 195
- to_param, 275
- ts:in, 44–46, 49
- ts:start, 45–46, 49
- ts:stop, 46, 49
- Twitter, 301, 309

U

- Uniqueness validations, 274
- unless, 135
- Unnested resources, 176–179
- Unprotected Jewels
 - init.rb, 252–253
 - plugin integration, 251–254
 - sqlite, 256, 260
 - test/factories.rb, 256
- Untested Rake
 - class method, 248–250
 - FakeWeb, 250
 - FileUtils::NoWrite, 250
 - forking, 247
 - mocking and stubbing, 247
 - rake tasks, 246–248
- Unused gems, 216

- up method, 292–296
- URL helpers, 100–101, 168, 255
- URL mapping, 163
- “Use only one dot,” 5
- User authentication plugins, 118–122
- User authorization code, 74–78
- users_attributes, 27–28
- UserController, 9–11, 92–93, 163–164, 167–168, 205
- UserSessionsController, 121–122
- users.yml, 223, 225, 229
- use_transactional_fixtures, 147

V

- #valid?, 242
- Validation macros, 53
- Validations, 25–28, 297–299, 307
- Vendor Junk Drawer, 216
- vendor/gems directory, 217
- vendor/plugins directory, 217
- version model, 130–142
- Versioning, 85, 217–219
- Versions, 128–129
- View helpers, 91–100, 103–106
- Views
 - ERb, 89, 91, 94–95
 - layer, 91
 - Markup Mayhem, 107–116
 - MIME, 89
 - PHPitis, 91–106
 - RJS, 89, 91–92
- Virtual Proxy, 37
- Vlissides, John M., 54, 225
- Voyeuristic models
 - Active Record associations, 3–7, 11–12
 - Active Record scopes, 10–13
 - delegate method, 6–7
 - encapsulation, 4–5
 - find() calls, 7–13
 - Law of Demeter, 3–7

`UserController`, 9–11
wrapper methods, 6

W

Wanstrath, Chris, 287
webrat, 264
Wet Validations, 297–299
`WHERE` clauses, 275
Whitespace sensitivity, 113–114, 116
`will_paginate` library, 44–45
Wrapper methods, 6
`written_at`, 126–128

X

Xapian, 43
XML, 17, 184, 198–199
XPath, 198–199
`xUnit` Pattern, 221

Y

YAML, 223–225
`yield`, 95–98
yum, 44