everyday **Rails**

Testing with RSpec

A practical approach to test-driven development

Aaron Sumner

Everyday Rails Testing with RSpec

A practical approach to test-driven development

Aaron Sumner

This book is for sale at http://leanpub.com/everydayrailsrspec

This version was published on 2017-07-11

Tweet This Book!

Please help Aaron Sumner by spreading the word about this book on Twitter!

The suggested hashtag for this book is #everydayrailsrspec.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#everydayrailsrspec

Contents

CONTENTS

Preface to this edition

Thanks for checking out this edition of *Everyday Rails Testing with RSpec*. It's been a long time coming, and a lot has changed. I hope you'll find your patience has been rewarded.

What took me so long? As I said, a lot has changed–both in terms of the book's contents, and the testing landscape in Rails in general. Let's talk about the latter first. With version 5.0, the Rails team announced that directly testing controllers is effectively deprecated. This was great news for me–even though the previous edition of this book had *three chapters* devoted to the practice, that was more a reflection on the time and effort it took me to understand it to begin with, and not so much how frequently I test at that level today.

One year later, Rails 5.1 is out and finally introduces the same high-level system testing I've always espoused in this book from the beginning, expressly built into the Rails framework. It's not RSpec, so it's not quite the same, but I'm happy that developers who want to stick with what Rails gives them out of the box can add tests at multiple levels of the application.

Meanwhile, RSpec marches on. There are lots of wonderful new features to make your tests more expressive. And a lot of us still prefer RSpec, and will take those few extra steps to add it to our apps.

That's what's new outside of my control. Now let's look at changes I've made to hopefully make this edition of *Everyday Rails Testing with RSpec* better than its predecessors. In many cases, these are changes I've wanted to make, regardless of what the Rails and RSpec teams have done with their respective frameworks.

This book started as a series of blog posts on Everyday Rails[1]. Five years ago, I began writing about my own experiences learning how to write tests for Rails applications. The blog posts proved to be among my more popular pieces, and I decided to collect them into a book, along with some exclusive content and a full code sample. In the

---

[1] https://everydayrails.com

time since then, the book has sold beyond my wildest expectations, and helped a lot of people facing those same struggles I had when I began to write it.

Funny thing about software, though. It's called *soft* for a reason. It changes. As we collectively and individually understand problems, we adapt our approaches to solving them. My techniques for writing tests today still have their roots in those early blog posts and the first editions of the book, but I've added to them, pruned from them, honed them.

The tricky part I've faced over the past year has been, *how do I synthesize that next level of learning into something that new developers can learn from, then add, prune, and hone to develop their own strategies?* Luckily, the original learning framework for the book still holds true: Start with a basic app, reasonably tested through a browser (or, these days, something like Postman for your APIs). Begin testing small blocks. Test the obvious. Build up to more complicated tests. Flip it around: Test first, code second. And over time, build your own strategies for effective testing.

Meanwhile, I've never been happy with the sample application used in previous editions. I liked that it was simple enough to keep the whole thing in a reader's head while also introducing testing concepts, but that simplicity got in my way at times–not enough code to add meaningful coverage. I resorted to modifying the same handful of files from chapter to chapter, which led to version control conflicts when I tried to apply even the simplest updates. The bigger (but still not too big) sample app in this edition has been much more pleasant to work with. It was also the first application I've written in years that *wasn't* written test-first!

Anyway, I hope you enjoy the book, whether you're new to testing, or are just curious about how my opinions on test-driven development and other testing practices have evolved since that first edition. While I've gone through the text and code multiple times to look for problems, you may come across something that's not quite right or that you'd do differently. If you find any errors or have suggestions, please share in the GitHub issues[2] for this release and I'll address them as promptly as possible.

Thanks to all of you for your support–hope you like this edition, and I hope to hear from you soon on GitHub, Twitter or email.

---

[2]https://github.com/everydayrails/everydayrails-rspec-2017/issues

1. Introduction

Ruby on Rails and automated testing go hand in hand. Rails ships with a built-in test framework. It automatically creates boilerplate test files when you use generators, ready for you to fill in with your own tests. Yet, many people developing in Rails are either not testing their projects at all, or at best, only adding a few token specs on basic things that may not even be useful, or informative.

In my opinion, there are several reasons for this. Perhaps working with Ruby or opinionated web frameworks is a novel enough concept, and adding an extra layer of work seems like just that–*extra work!* Or maybe there is a perceived time constraint–spending time on writing tests takes time away from writing the features our clients or bosses demand. Or maybe the habit of defining *test* as the practice of clicking links in the browser is just too hard to break.

I've been there. I've never considered myself an engineer in the traditional sense–yet just like traditional engineers, I have problems to solve. And, typically, I find solutions to these problems in building software. I've been developing web applications since 1995. For a long time, I worked as a solo developer on shoestring, public sector projects. Aside from some structured exposure to BASIC as a kid, a little C++ in college, and a wasted week of Java training in my second grown-up job outside of college, I've never had any honest-to-goodness schooling in software development. In fact, it wasn't until 2005, when I'd had enough of hacking ugly spaghetti-style[3] PHP code, that I sought out a better way to write web applications.

I'd looked at Ruby before, but never had a serious use for it until Rails began gaining steam. There was a lot to learn–a new language, an actual *architecture*, and a more object-oriented approach. (Despite what you may think about Rails' treatment of object orientation, it's far more object oriented than anything I wrote in my pre-framework days.) Even with all those new challenges, though, I was able to create complex applications in a fraction of the time it took me in my previous, framework-less efforts. I was hooked.

---

[3]http://en.wikipedia.org/wiki/Spaghetti_code

That said, early Rails books and tutorials focused more on development speed (build a blog in 15 minutes!) than on good practices like testing. If testing were covered at all, it was generally reserved for a chapter toward the end. Now, to be fair, newer educational resources on Rails have addressed this shortcoming, and demonstrate how to test applications from the beginning. In addition, a number of books have been written *specifically* on the topic of testing. But without a sound approach to the testing side, many developers–especially those in a similar position to the one I was in–may find themselves without a consistent testing strategy. If there are any tests at all, they may not be reliable, or meaningful. These tests don't lead to *developer confidence.*

My first goal with this book is to introduce you to a consistent strategy that works for *me*–one that you can then, hopefully, adapt to make work consistently for *you,* too. If I'm successful, then by reading this book, you'll *test with confidence.* You'll be able to make changes to your code, knowing that your test suite has your back and will let you know if something breaks.

Why RSpec?

To be clear, I have nothing against other Ruby testing frameworks. If I'm writing a standalone Ruby library, I usually rely on MiniTest. I stick with RSpec when it comes to developing and testing my Rails applications, though.

Maybe it stems from my backgrounds in copywriting and software development, but for me, RSpec's capacity for specs that are readable, without being cumbersome, is a winner. I'll talk more about this later in the book, but I've found that with a little coaching, even most non-technical people can read a spec written in RSpec and understand what's going on. It's expressive in such a way that using RSpec to describe how I expect my software to behave has become second nature. The syntax flows from my fingers, and is pleasant to read in the future when I'm making changes to my software.

My second goal with this book is to help you feel comfortable with the RSpec functionality and syntax you're most likely to use on a regular basis. RSpec is a complex framework, but like many complex systems, you'll likely find yourself using 20 percent of the available functionality for 80 percent of your work. With that in

mind, this is not a complete guide to RSpec or companion libraries like Capybara, but instead focuses on the tools I've used for years to test my own Rails applications. It will also introduce you to common patterns so that, when you run into an issue that's not covered in the book, you'll be able to intelligently look for solutions and not get stuck.

Who should read this book

If Rails is your first foray into a web application framework, and your past programming experience didn't involve any testing to speak of, this book will hopefully help you get started. If you're *really* new to Rails, you may find it beneficial to review coverage of development and basic testing in the likes of Michael Hartl's *Rails Tutorial*, Daniel Kehoe's *Learn Ruby on Rails*, or Sam Ruby's *Agile Web Development with Rails 5*, before digging into *Testing Rails with RSpec*. My book assumes you've got some basic Rails skills under your belt. In other words, it won't teach you how to use Rails, and it won't provide a ground-up introduction to the testing tools built into the framework. Instead, we're going to be installing RSpec and a few extras to make the testing process as easy as possible to comprehend and manage. So if you're new to Rails, check out one of those resources first, then come back to this book.

> Refer to *More Testing Resources for Rails* at the end of this book for links to these and other books, websites, and testing tutorials.

If you've been developing in Rails for a little while, but testing is still a foreign concept, then this book is for you! I was in your position for a long time, and the techniques I'll share here helped me improve my test coverage and think more like a test-driven developer. I hope they'll do the same for you.

Specifically, you should probably have a grasp of

- Server-side Model-View-Controller application conventions, as used in Rails
- Bundler for gem dependency management
- How to work with the Rails command line
- Enough Git to switch between branches of a repository

If you're already familiar with using Test::Unit, MiniTest, or even RSpec itself, and already have a workflow in place that gives you confidence, you may be able to fine-tune some of your approach to testing your applications. I hope you'll also learn from my opinionated approach to testing, and how to go from testing your code to testing with purpose.

This is *not* a book on general testing theory, and it doesn't dig too deeply into performance issues that can creep into legacy software over time. Other books may be of more use on those topics. Refer to *More Testing Resources for Rails* at the end of this book for links to these and other books, websites, and testing tutorials.

My testing philosophy

What kind of testing is best–unit tests, or integration? Should I practice test-driven development, or behavior-driven development (and what's the difference, anyway)? Should I write my tests before I write code, or after? Or should I even bother to write tests at all?

Discussing the *right* way to test your Rails application can invoke major shouting matches amongst programmers–not quite as bad as, say, the Mac versus PC or Vim versus Emacs debates, but still not something to bring up in an otherwise pleasant conversation with fellow Rubyists. In fact, David Heinemeier-Hansen's keynote at Railsconf 2014, in which he declared TDD as "dead,"[4] recently sparked a fresh round of debates on the topic.

Yes, there is a right way to do testing–but if you ask me, there are degrees of *right* when it comes to testing. My approach focuses on the following basic beliefs:

- Tests should be reliable.
- Tests should be easy to write.
- Tests should be easy to understand–today, and in the future.

In summary: Tests should give you *confidence* as a software developer. If you mind these three factors in your approach, you'll go a long way toward having a sound

---

[4]http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

test suite for your application–not to mention becoming an honest-to-goodness practitioner of Test-Driven Development.

Yes, there are some tradeoffs–in particular:

- We're not focusing on speed, though we will talk about it later.
- We're not focusing on overly DRY code in our tests. But in tests, that's not necessarily a bad thing. We'll talk about this, too.

In the end, though, the most important thing is that *you'll have good tests*–and reliable, understandable tests are a great way to start, even if they're not quite as optimized as they could be. This is the approach that finally got me over the hump between writing a lot of application code, calling a round of browser-clicking "testing," and hoping for the best; versus taking advantage of a fully automated test suite and using tests to drive development and ferret out potential bugs and edge cases.

And that's the approach we'll take in this book.

How the book is organized

In *Testing Rails with RSpec* I'll walk you through taking a basic Rails application from completely untested to respectably tested with RSpec. The book covers Rails 5.1 and RSpec 3.6, which are the respective current versions of each as of this writing.

The book is organized into the following activities:

- You're reading chapter 1, *Introduction*, now.
- In chapter 2, *Setting Up RSpec*, we'll set up a new or existing Rails application to use RSpec.
- In chapter 3, *Model Specs*, we'll tackle testing our application's models through simple, yet reliable, unit testing.
- Chapter 4, *Creating Meaningful Test Data*, looks at techniques for generating test data.
- In chapter 5, *Controller Specs*, we'll write tests directly against our app's controllers.

- In chapter 6, *Feature Specs*, we'll move on to integration testing with feature specs, thus testing how the different parts of our application interact with one another.
- In chapter 7, *Request Specs*, we'll look at how to test your application's programmatic interface directly, without going through a traditional web user interface.
- In chapter 8, *DRY Specs*, we'll explore when and how to reduce duplication in tests–and when to leave things alone.
- In chapter 9, *Testing in Isolation*, we'll cover techniques for writing focused tests for fast feedback.
- Chapter 10, *Testing the Rest*, covers testing those parts of our code we haven't covered yet–things like email, file uploads, time-specific functionality, and external web services.
- I'll go through a step-by-step demonstration of test-driven development in chapter 11, *Toward Test-driven Development*.
- Finally, we'll wrap things up in chapter 12, *Parting Advice*.

Each chapter contains the step-by-step process I used to get better at testing my own software. Many chapters conclude with a questions section, designed to encourage further thinking into the *how* and *why* of testing, followed by a few exercises to follow when using these techniques on your own. Again, I strongly recommend working through the exercises in your own applications–it's one thing to follow along with a tutorial; it's another thing entirely to apply what you learn to your own situation. We won't be building an application together in this book, just exploring code patterns and techniques. Take those techniques and make your own projects better!

Downloading the sample code

Speaking of the sample code, you can find a completely tested application on GitHub.

Get the source!

https://github.com/everydayrails/everydayrails-rspec-2017

If you're familiar with Git (and, as a Rails developer, you should be), you can clone the source to your computer. Each chapter's work has its own branch. Grab that chapter's source to see the completed code, or the previous chapter's source if you'd like to follow along with the book. Branches are labeled by chapter number. I'll also tell you which branch to check out at the start of that chapter, and provide a link to show all changes between the current and previous chapters.

This book is structured so that each chapter builds on the previous one. That way, you can use the previous chapter's branch as a starting point for a current chapter. For example, if I wanted to follow along with chapter 5's code, I could start from chapter 4:

```
$ git checkout -b my-05-controllers origin/04-factories
```

If you're not familiar with Git, you may still download the sample code a given chapter. To begin, open the project on GitHub. Then, locate the branch selector and select that chapter's branch:
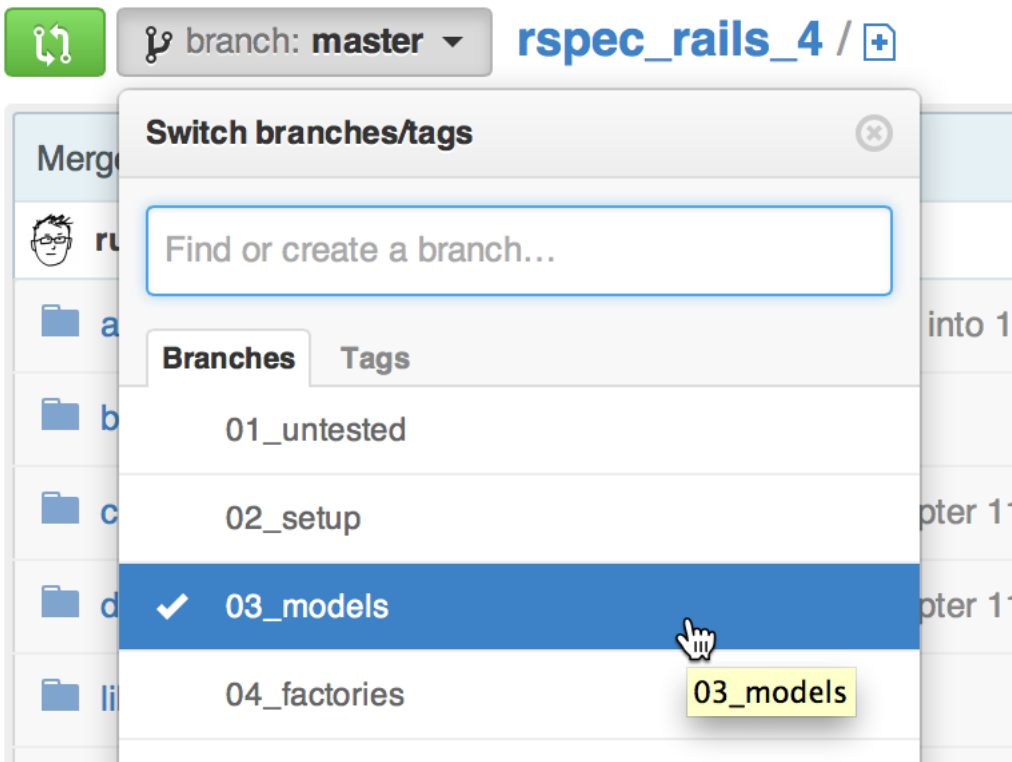
Finally, click the ZIP download button to save the source to your computer:

SSH clone URL

git@github.com:eve

You can clone with HTTPS, SSH, or Subversion. ⑦

🖥 **Clone in Desktop**

☁ **Download ZIP**

🖑

Download this repository as a zip file

🔑 Git Immersion[5] is an excellent, hands-on way to learn the basics of Git on the command line. So is Try Git[6].

Code conventions

I'm using the following setup for this application:

- **Rails 5.1**: The latest version of Rails is the big focus of this book. As far as I know, most of the techniques I'm using will apply to any version of Rails from 3.0 onward. Your mileage may vary with some of the code samples, but I'll do my best to let you know where things might differ.
- **Ruby 2.4**: Rails 5.0 requires a minimum of Ruby 2.2. If you're adding tests to an app in an older version of Rails, you shouldn't run into any major challenges in Ruby 2.0, 2.1, or 2.2.

---

[5]http://gitimmersion.com/
[6]http://try.github.io

- **RSpec 3.6:** RSpec 3.0 was released in spring, 2014. RSpec 3.6 was released to coincide with Rails 5.1, and is mostly compatible with the 3.0 release. It's relatively close in syntax to RSpec 2.14, though there are a few differences.

If something's particular to these versions, I'll do my best to point it out. If you're working from an older version of Rails, RSpec, or Ruby, previous versions of the book are available as free downloads through Leanpub with your paid purchase of this edition. They're not feature-for-feature identical, but you should hopefully be able to see some of the basic differences.

Again, **this book is not a traditional tutorial!** The code provided here isn't intended to walk you through building an application. It's here to help you understand and learn testing patterns and habits to apply to your own Rails applications. In other words, you can copy and paste, but it's probably not going to do you a lot of good. You may be familiar with this technique from Zed Shaw's Learn Code the Hard Way series[7].

Everyday Rails Testing Rails with RSpec is not in that exact style, but I do agree with Zed that typing things yourself, as opposed to copying-and-pasting from Stack Overflow or an ebook, is a better way to learn.

Discussion and errata

I've put a lot of time and effort into making sure *Testing Rails with RSpec* is as error-free as possible, but you may find something I've missed. If that's the case, head on over to the issues section for the source on GitHub to share an error or ask for more details: https://github.com/everydayrails/everydayrails-rspec-2017/issues

A note about gem versions

The gem versions used in this book and the sample application are current as I write this RSpec 3.6/Rails 5.1 edition, in spring, 2017. Of course, any and all may update frequently, so keep tabs on them on Rubygems.org, GitHub, and your favorite Ruby news feeds for updates.

---

[7]http://learncodethehardway.org/

About the sample application

Our sample application is a project management application. While it's not as powerful or polished as Trello or Basecamp, it's got just enough features to get started with testing.

To start, the application supports the following features:

- A user can add a project, visible only to her.
- A user can add tasks, notes, and attachments to a project.
- A user can mark tasks as complete.
- A user' account has an avatar, provided by the Gravatar service.
- A developer can access a public API to develop external client applications.

Up to this point, I've been intentionally lazy and only used Rails' default generators to create the entire application (see the *01-untested*[8] branch of the sample code). This means I have a *test* directory full of untouched test files and fixtures. I could run `bin/rails test` at this point, and perhaps some of these tests would even pass. But since this is a book about RSpec, we'll delete this folder, set up Rails to use RSpec instead, and build out a reliable test suite. That's what we'll walk through in this book.

First things first: We need to configure the application to recognize and use RSpec. Let's get started!

---

[8] https://github.com/everydayrails/everydayrails-rspec-2017/tree/01-untested

3. Model specs

With RSpec successfully installed, we can now put it to work and begin building a suite of reliable tests. We'll get started with the app's core building blocks–its models.

In this chapter, we'll complete the following tasks:

- First we'll create model specs for existing models.
- Then, we'll write passing tests for a model's validations, class, and instance methods, and organize our specs in the process.

We'll create our first spec files for existing models manually. Later, when adding new models to the application, the handy RSpec generators we configured in chapter 2 will generate placeholder files for us.

> You can view all the code changes for this chapter in a single diff[9] on GitHub.
>
> If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:
>
> ```
> git checkout -b my-03-models origin/02-setup
> ```

Anatomy of a model spec

I think it's easiest to learn testing at the model level, because doing so allows you to examine and test the core building blocks of an application. Well-tested code at this level provides a solid foundation for a reliable overall code base.

To get started, a model spec should include tests for the following:

---

[9]https://github.com/everydayrails/everydayrails-rspec-2017/compare/02-setup...03-models

- When instantiated with valid attributes, a model should be valid.
- Data that fail validations should not be valid.
- Class and instance methods perform as expected.

This is a good time to look at the basic structure of an RSpec model spec. I find it helpful to think of them as individual outlines. For example, let's look at our User model's requirements:

```
describe User do
  it "is valid with a first name, last name, email, and password"
  it "is invalid without a first name"
  it "is invalid without a last name"
  it "is invalid without an email address"
  it "is invalid with a duplicate email address"
  it "returns a user's full name as a string"
end
```

We'll expand this outline in a few minutes, but this gives a lot to work with for starters. It's a simple spec for an admittedly simple model, but points to our first four best practices:

- **It describes a set of expectations**–in this case, what the User model should look like, and how it should behave.
- **Each example (a line beginning with `it`) only expects one thing**. Notice that I'm testing the `first_name`, `last_name`, and `email` validations separately. This way, if an example fails, I know it's because of that *specific* validation, and I don't have to dig through RSpec's output for clues–at least, not as deeply.
- **Each example is explicit**. The descriptive string after `it` is technically optional in RSpec. However, omitting it makes your specs more difficult to read.
- **Each example's description begins with a verb**, **not should**. Read the expectations aloud: *User is invalid without a first name, User is invalid without last name, User returns a user's full name as a string*. Readability is important, and a key feature of RSpec!

With these best practices in mind, let's build a spec for the *User* model.

Creating a model spec

In chapter 2, we set up RSpec to automatically generate boilerplate test files whenever we add new models and controllers to the application. We can invoke generators anytime, though. Here, we'll use one to generate a starter file for our first model spec.

Begin by using the *rspec:model* generator on the command line:

```
1    $bin/rails g rspec:model user
```

RSpec reports that a new file was created:

```
1    Running via Spring preloader in process 32008
2          create  spec/models/user_spec.rb
```

Let's open the new file and take a look.

spec/models/user_spec.rb

```
1    require 'rails_helper'
2
3    RSpec.describe User, type: :model do
4      pending "add some examples to (or delete) #{__FILE__}"
5    end
```

The new file gives us our first look at some RSpec syntax and conventions. First, we *require* the file *rails_helper* in this file, and will do so in pretty much every file in our test suite. This tells RSpec that we need the Rails application to load in order to run the tests contained in the file. Next, we're using the *describe* method to list out a set of things a *model* named *User* is expected to do. We'll talk more about *pending* in chapter 11, when we begin practice test-driven development. For now, happens if we run this, using `bin/rspec`?

```
Running via Spring preloader in process 41653

User
  add some examples to (or delete)
  /Users/asumner/code/examples/projects/spec/models/user_spec.rb
  (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User add some examples to (or delete)
  /Users/asumner/code/examples/projects/spec/models/user_spec.rb
     # Not yet implemented
     # ./spec/models/user_spec.rb:4


Finished in 0.00107 seconds (files took 0.43352 seconds to load)
1 example, 0 failures, 1 pending
```

> You don't need to use generators to create spec files, but they're a good way to
> prevent silly errors caused by typos.

Let's keep the *describe* wrapper, but replace its contents with the outline we created
a few minutes ago:

spec/models/user_spec.rb

```
1   require 'rails_helper'
2
3   RSpec.describe User, type: :model do
4     it "is valid with a first name, last name, email, and password"
5     it "is invalid without a first name"
6     it "is invalid without a last name"
7     it "is invalid without an email address"
8     it "is invalid with a duplicate email address"
9     it "returns a user's full name as a string"
10  end
```

We'll fill in the details in a moment, but if we ran the specs right now from the command line (by typing `bin/rspec` on the command line) the output would be similar to the following:

```
Running via Spring preloader in process 32556

User
  is valid with a first name, last name, email, and password (PENDING:
  Not yet implemented)
  is invalid without a first name (PENDING: Not yet implemented)
  is invalid without a last name (PENDING: Not yet implemented)
  is invalid without an email address (PENDING: Not yet implemented)
  is invalid with a duplicate email address (PENDING: Not yet implemented)
  returns a user's full name as a string (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User is valid with a first name, last name, email, and password
     # Not yet implemented
     # ./spec/models/user_spec.rb:4

  2) User is invalid without a first name
     # Not yet implemented
```

```
  # ./spec/models/user_spec.rb:5

3) User is invalid without a last name
   # Not yet implemented
   # ./spec/models/user_spec.rb:6

4) User is invalid without an email address
   # Not yet implemented
   # ./spec/models/user_spec.rb:7

5) User is invalid with a duplicate email address
   # Not yet implemented
   # ./spec/models/user_spec.rb:8

6) User returns a user's full name as a string
   # Not yet implemented
   # ./spec/models/user_spec.rb:9
```

```
Finished in 0.00176 seconds (files took 2.18 seconds to load)
6 examples, 0 failures, 6 pending
```

Great! Six pending specs. RSpec marks them as *pending* because we haven't written any actual code to perform the tests. Let's do that now, starting with the first example.

> Older versions of Rails required you to manually copy your development database structure into your test database via a Rake task. Now, however, Rails handles this for you automatically anytime you run a migration– most of the time, anyway. If you get an error suggesting that migrations are missing in your test environment, get them up to date by running `bin/rails db:migrate RAILS_ENV=test`.

The RSpec syntax

In 2012, the RSpec team announced a new, preferred alternative to the traditional `should`, added to version 2.11. Of course, this happened just a few days after I released

the first complete version of this book–it can be tough to keep up with this stuff sometimes!

This new approach alleviates some technical issues caused by the old should syntax[10]. Instead of saying something should or should_not match expected output, you expect something to or not_to be something else.

As an example, let's look at this sample test, or *expectation*. In this example, 2 + 1 should always equal 3, right? In the old RSpec syntax, this would be written like this:

```
it "adds 2 and 1 to make 3" do
  (2 + 1).should eq 3
end
```

The new syntax passes the test value into an expect() method, then chains a matcher to it:

```
it "adds 2 and 1 to make 3" do
  expect(2 + 1).to eq 3
end
```

If you're searching Google or Stack Overflow for help with an RSpec question, or are working with an older Rails application, there's still a good chance you'll find information using the old should syntax. This syntax still technically works in current versions of RSpec, but you'll get a deprecation warning when you try to use it. You *can* configure RSpec to turn off these warnings, but in all honesty, you're better off learning to use the preferred expect() syntax.

So what does that syntax look like in a real example? Let's fill out that first expectation from our spec for the User model:

---

[10]http://myronmars.to/n/dev-blog/2012/06/rspecs-new-expectation-syntax

spec/models/user_spec.rb

```ruby
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    it "is valid with a first name, last name, email, and password" do
5      user = User.new(
6        first_name: "Aaron",
7        last_name:  "Sumner",
8        email:      "tester@example.com",
9        password:   "dottle-nouveau-pavilion-tights-furze",
10     )
11     expect(user).to be_valid
12   end
13
14   it "is invalid without a first name"
15   it "is invalid without a last name"
16   it "is invalid without an email address"
17   it "is invalid with a duplicate email address"
18   it "returns a user's full name as a string"
19 end
```

This simple example uses an RSpec *matcher* called be_valid to verify that our model knows what it has to look like to be valid. We set up an object (in this case, a new-but-unsaved instance of User called user), then pass that to expect to compare to the matcher.

Now, if we run bin/rspec from the command line again, we see one passing example:

```
Running via Spring preloader in process 32678

User
  is valid with a first name, last name and email, and password
  is invalid without a first name (PENDING: Not yet implemented)
  is invalid without a last name (PENDING: Not yet implemented)
  is invalid without an email address (PENDING: Not yet implemented)
  is invalid with a duplicate email address (PENDING: Not yet implemented)
  returns a user's full name as a string (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your suite's s\
tatus)

  1) User is invalid without a first name
     # Not yet implemented
     # ./spec/models/user_spec.rb:14

  2) User is invalid without a last name
     # Not yet implemented
     # ./spec/models/user_spec.rb:15

  3) User is invalid without an email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:16

  4) User is invalid with a duplicate email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:17

  5) User returns a user's full name as a string
     # Not yet implemented
     # ./spec/models/user_spec.rb:18



Finished in 0.02839 seconds (files took 0.28886 seconds to load)
6 examples, 0 failures, 5 pending
```

Congratulations, you've written your first test! Now let's get into testing more of our

code, so we don't have any more pending tests.

Testing validations

Validations are a good way to get comfortable with automated testing. These tests
can usually be written in just a line or two of code. Let's look at some detail to our
`first_name` validation spec:

spec/models/user_spec.rb

```
1  it "is invalid without a first name" do
2    user = User.new(first_name: nil)
3    user.valid?
4    expect(user.errors[:first_name]).to include("can't be blank")
5  end
```

This time, we *expect* that when we call the `valid?` method on the new user (with a
`first_name` explicitly set to `nil`), we'll find it *not* be valid, thus returning the shown
error message on the user's `first_name` attribute. We check for this using RSpec's
`include` matcher, which checks to see if a value is included in an enumerable value.
And when we run RSpec again, we should be up to two passing specs.

There's a small problem in our approach so far. We've got a couple of passing tests,
but we never saw them *fail*. This can be a warning sign, especially when starting
out. We need to be certain that the test code is doing what it's intended to do, also
known as *exercising the code under test*.

There are a couple of things we can do to prove that we're not getting false positives.
First, let's flip that expectation by changing `to` to `to_not`:

spec/models/user_spec.rb

```
1  it "is invalid without a first name" do
2    user = User.new(first_name: nil)
3    user.valid?
4    expect(user.errors[:first_name]).to_not include("can't be blank")
5  end
```

And sure enough, RSpec reports a failure:

```
Failures:

  1) User is invalid without a first name
     Failure/Error: expect(user.errors[:first_name]).to_not
     include("can't be blank")
       expected ["can't be blank"] not to include "can't be blank"
     # ./spec/models/user_spec.rb:17:in `block (2 levels) in <top \
     (required)>'
     # /Users/asumner/.rvm/gems/ruby-2.4.1/gems/spring-commands-rspec-\
     1.0.4/lib/spring/commands/rspec.rb:18:in `call'
     # -e:1:in `<main>'

Finished in 0.06211 seconds (files took 0.28541 seconds to load)
6 examples, 1 failure, 4 pending

Failed examples:

rspec ./spec/models/user_spec.rb:14 # User is invalid without a first name
```

> RSpec provides to_not and not_to for these types of expectations. They're
> interchangeable. I use to_not in the book, since that's what is commonly
> used in RSpec's documentation.

We can also modify the application code, to see how it affects the test. Undo the
change we just made to the test (switch to_not back to to), then open the User
model and comment out the first_name validation:

app/models/user.rb

```
1   class User < ApplicationRecord
2     # Include default devise modules. Others available are:
3     # :confirmable, :lockable, :timeoutable and :omniauthable
4     devise :database_authenticatable, :registerable,
5            :recoverable, :rememberable, :trackable, :validatable
6
7     # validates :first_name, presence: true
8     validates :last_name, presence: true
9
10    # rest of file omitted ...
```

Run the specs again, and you should again see a failure–we told RSpec that a user with no first name should be invalid, but our application code didn't support that.

These are easy ways to verify your tests are working as expected, especially as you progress from testing simple validations to more complex logic, and are testing code that's already been written. If you don't see a change in test output, then there's a good chance that the test is not actually interacting with the code, or that the code behaves differently than you expect.

Now we can use the same approach to test the :last_name validation.

spec/models/user_spec.rb

```
1   it "is invalid without a last name" do
2     user = User.new(last_name: nil)
3     user.valid?
4     expect(user.errors[:last_name]).to include("can't be blank")
5   end
```

You may be thinking that these tests are relatively pointless–how hard is it to make sure validations are included in a model? The truth is, they can be easier to omit than you might imagine. More importantly, though, if you think about what validations your model should have *while* writing tests (ideally, and eventually, in a Test-Driven Development style of coding), you are more likely to remember to include them.

Let's build on our knowledge so far to write a slightly more complicated test–this time, to check the uniqueness validation on the email attribute:

spec/models/user_spec.rb

```
1   it "is invalid with a duplicate email address" do
2     User.create(
3       first_name:  "Joe",
4       last_name:   "Tester",
5       email:       "tester@example.com",
6       password:    "dottle-nouveau-pavilion-tights-furze",
7     )
8     user = User.new(
9       first_name:  "Jane",
10      last_name:   "Tester",
11      email:       "tester@example.com",
12      password:    "dottle-nouveau-pavilion-tights-furze",
13    )
14    user.valid?
15    expect(user.errors[:email]).to include("has already been taken")
16   end
```

Notice a subtle difference here: In this case, we persisted a user (calling `create` on `User` instead of `new`) to test against, then instantiated a second user as the subject of the actual test. This, of course, requires that the first, persisted user is valid (with a first, last name, email, and password) and has the same email address assigned to it. In chapter 4, we'll look at utilities to streamline this process. In the meantime, run `bin/rspec` to see the new test's output.

Now let's test a more complex validation. To do so, we'll set aside tests for the User model, and turn to Projects. Say we want to make sure that users can't give two of their projects the same name–the name should be unique within the scope of that user. In other words, I can't have two projects named *Paint the house*, but you and I could each have our own project named *Paint the house*. How might you test that?

We'll start by creating a new spec file for the Project model:

```
bin/rails g rspec:model project
```

Next, add two examples to the new file. We'll test that a single user can't have two projects with the same name, but two different users can each have a project with the same name.

spec/models/project_spec.rb

```
1   require 'rails_helper'
2
3   RSpec.describe Project, type: :model do
4     it "does not allow duplicate project names per user" do
5       user = User.create(
6         first_name: "Joe",
7         last_name:  "Tester",
8         email:      "joetester@example.com",
9         password:   "dottle-nouveau-pavilion-tights-furze",
10      )
11
12      user.projects.create(
13        name: "Test Project",
14      )
15
16      new_project = user.projects.build(
17        name: "Test Project",
18      )
19
20      new_project.valid?
21      expect(new_project.errors[:name]).to include("has already been taken")
22    end
23
24    it "allows two users to share a project name" do
25      user = User.create(
26        first_name: "Joe",
27        last_name:  "Tester",
28        email:      "joetester@example.com",
29        password:   "dottle-nouveau-pavilion-tights-furze",
30      )
31
32      user.projects.create(
33        name: "Test Project",
34      )
35
36      other_user = User.create(
```

```
37          first_name: "Jane",
38          last_name:  "Tester",
39          email:      "janetester@example.com",
40          password:   "dottle-nouveau-pavilion-tights-furze",
41        )
42
43      other_project = other_user.projects.build(
44        name: "Test Project",
45        )
46
47      expect(other_project).to be_valid
48    end
49  end
```

This time, since the `User` and `Project` models are coupled via an Active Record relationship, we need to provide a little extra information. In the case of the first example, we've got a user to which both projects are assigned. In the second, the same project name is assigned to two unique projects, belonging to unique users. Note that, in both examples, we have to `create` the users, or persist them in the database, in order to assign them to the projects we're testing.

And since the `Project` model has the following validation:

app/models/project.rb

```
validates :name, presence: true, uniqueness: { scope: :user_id }
```

These new specs will pass without issue. Don't forget to check your work–try temporarily commenting out the validation, or changing the tests so they expect something different. Do they fail now?

Of course, validations can be more complicated than just requiring a specific scope. Yours might involve a complex regular expression, or a custom validator. Get in the habit of testing these validations–not just the happy paths where everything is valid, but also error conditions. For instance, in the examples we've created so far, we tested what happens when an object is initialized with `nil` values. If you have a validation to ensure that an attribute must be a number, try sending it a string. If your validation requires a string to be four-to-eight characters long, try sending it three characters, and nine.

Testing instance methods

Let's resume testing the User model now. In our app, it would be convenient to only have to refer to `@user.name` to render our users' full names instead of concatenating the first and last names into a new string every time. To handle this, we've got this method in the User class:

app/models/user.rb

```ruby
def name
  [firstname, lastname].join(' ')
end
```

We can use the same basic techniques we used for our validation examples to create a passing example of this feature:

spec/models/user_spec.rb

```ruby
1  it "returns a user's full name as a string" do
2    user = User.new(
3      first_name: "John",
4      last_name:  "Doe",
5      email:      "johndoe@example.com",
6    )
7    expect(user.name).to eq "John Doe"
8  end
```

> RSpec requires `eq` or `eql`, not `==`, to indicate an expectation of equality.

Create test data, then tell RSpec how you expect it to behave. Easy, right? Let's keep going.

Testing class methods and scopes

We've got some simple functionality to search notes for a given term. For the sake of demonstration, it's currently implemented as a scope on the Note model:

app/models/note.rb

```
1  scope :search, ->(term) {
2    where("LOWER(message) LIKE ?", "%#{term.downcase}%")
3  }
```

Let's add a third file to our growing test suite for the Note model. After creating it
with the rspec:model generator, add an initial test:

spec/models/note_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe Note, type: :model do
4    it "returns notes that match the search term" do
5      user = User.create(
6        first_name: "Joe",
7        last_name:  "Tester",
8        email:      "joetester@example.com",
9        password:   "dottle-nouveau-pavilion-tights-furze",
10     )
11
12     project = user.projects.create(
13       name: "Test Project",
14     )
15
16     note1 = project.notes.create(
17       message: "This is the first note.",
18       user: user,
19     )
20     note2 = project.notes.create(
21       message: "This is the second note.",
22       user: user,
23     )
24     note3 = project.notes.create(
25       message: "First, preheat the oven.",
26       user: user,
27     )
```

```
28
29      expect(Note.search("first")).to include(note1, note3)
30      expect(Note.search("first")).to_not include(note2)
31    end
32 end
```

The *search* scope should return a collection of notes matching the search term, and that collection should only include those notes–not ones that don't contain the term.

This test gives us some other things to experiment with: What happens if we flip around the to and to_not variations on the tests? Or add more notes containing the search term?

Testing for failures

We've tested the happy path–a user searches a term for which we can return results– but what about occasions when the search returns no results? We'd better test that, too. The following spec should do it:

spec/models/note_spec.rb

```
1 require 'rails_helper'
2
3 RSpec.describe Note, type: :model do
4   # search results spec ...
5
6   it "returns an empty collection when no results are found" do
7     user = User.create(
8       first_name: "Joe",
9       last_name:  "Tester",
10      email:      "joetester@example.com",
11      password:   "dottle-nouveau-pavilion-tights-furze",
12    )
13
14    project = user.projects.create(
15      name: "Test Project",
16    )
```

```
17
18       note1 = project.notes.create(
19         message: "This is the first note.",
20         user: user,
21       )
22       note2 = project.notes.create(
23         message: "This is the second note.",
24         user: user,
25       )
26       note3 = project.notes.create(
27         message: "First, preheat the oven.",
28         user: user,
29       )
30
31       expect(Note.search("message")).to be_empty
32     end
33   end
```

This spec uses RSpec's `be_empty` matcher to check if the array returned by `Note.search("message"` has anything in it–and since it's empty, it passes! We're testing not just for ideal results–the user searches for a term with results–but also for searches with no results.

More about matchers

We've already seen four matchers in action: `be_valid`, `eq`, `include`, and `be_empty`. First we used `be_valid`, which is provided by the *rspec-rails* gem to test a Rails model's validity. `eq` and `include` come from *rspec-expectations*, installed alongside *rspec-rails* when we set up our app to use RSpec in the previous chapter.

A complete list of RSpec's default matchers may be found in the *README* for the rspec-expectations repository on GitHub[11]. We'll look at several of these throughout this book. In chapter 8, we'll take a look at creating custom matchers of our own.

---

[11]https://github.com/rspec/rspec-expectations

DRYer specs with describe, context, before and after

So far, the specs we've created for notes have some redundancy: We create the same four objects in each example. Just as in your application code, the DRY principle applies to your tests (with some exceptions, which I'll talk about momentarily). Let's use a few more RSpec features to clean things up.

Focusing on the specs we just created for the Note model, the first thing I'm going to do is create a describe block *within* the describe Note block, to focus on the search feature. The general outline will look like this:

spec/models/note_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe Note, type: :model do
4
5    # validation specs
6
7    describe "search message for a term" do
8      # searching examples ...
9    end
10 end
```

Let's break things down even further by including a couple of context blocks–one for when we find a match, and one when no match is found:

spec/models/user_spec.rb

```ruby
1   require 'rails_helper'
2
3   RSpec.describe Note, type: :model do
4
5     # any other specs
6
7     describe "search message for a term" do
8
9       context "when a match is found" do
10        # matching examples ...
11      end
12
13      context "when no match is found" do
14        # non-matching examples ...
15      end
16    end
17  end
```

> 🔑 While describe and context are technically interchangeable, I prefer to
> use them like this–specifically, describe outlines general functionality of
> my class or feature; context outlines a specific state. In this case, we have
> a state of a search term with matching results selected, and a state with a
> non-matching search term selected.

As you may be able to spot, we're creating an outline of examples here to help us
sort similar examples together. This makes for a more readable spec. Now let's finish
cleaning up our reorganized spec with the help of a before hook. Code inside a
before block is run before code inside individual tests. It's also scoped within a
describe or context block–so in this example, the code in before will run prior to all
tests inside the "search message for a term" block, but not before other examples
outside of the new describe block.

spec/models/note_spec.rb

```ruby
 1  require 'rails_helper'
 2
 3  RSpec.describe Note, type: :model do
 4
 5    before do
 6      # set up test data for all tests in the file
 7    end
 8
 9    # validation tests
10
11    describe "search message for a term" do
12
13      before do
14        # set up extra test data for all tests related to search
15      end
16
17      context "when a match is found" do
18        # matching examples ...
19      end
20
21      context "when no match is found" do
22        # non-matching examples ...
23      end
24    end
25  end
```

RSpec's `before` hooks are a good way to start recognizing and cleaning up redundancy in your specs. There are other techniques for tidying up redundant test code, but `before` is probably the most common. A `before` block may be run once per example, once per block of examples, or once per run of the entire test suite:

- `before(:each)` runs before *each* test in a `describe` or `context` block. You can use the alias `before(:example)`, if you prefer, or just `before` as shown in this sample. So if a spec file has four tests in it, the `before` code would run four times.

- `before(:all)` runs once before *all* tests in a `describe` or `context` block. This is aliased as `before(:context)`. This time, the `before` code would run just once, then our four tests would run.
- `before(:suite)` runs before the entire suite of tests, across all files.

`before(:all)` and `before(:suite)` can help speed up test runs by isolating expensive test setup to a single run, but they can also lead to pollution across tests. Stick to using `before(:each)` whenever possible.

> If you define a `before` block as we have here, the code inside the block will run prior to *each* test. This can be made explicit by calling it with `before :each`. Use whichever style you and your team prefer.

If a spec requires some sort of post-example teardown, like disconnecting from an external service, we can also use an `after` hook to clean up after the examples. `after` has the same `each`, `all`, and `suite` options as `before`. Since RSpec handles cleaning up the database by default, I rarely use `after`.

Okay, let's see that full, organized spec:

spec/models/note_spec.rb

```ruby
 1  require 'rails_helper'
 2
 3  RSpec.describe Note, type: :model do
 4    before do
 5      @user = User.create(
 6        first_name: "Joe",
 7        last_name:  "Tester",
 8        email:      "joetester@example.com",
 9        password:   "dottle-nouveau-pavilion-tights-furze",
10      )
11
12      @project = @user.projects.create(
13        name: "Test Project",
14      )
15    end
```

```
16
17    it "is valid with a user, project, and message" do
18      note = Note.new(
19        message: "This is a sample note.",
20        user: @user,
21        project: @project,
22      )
23      expect(note).to be_valid
24    end
25
26    it "is invalid without a message" do
27      note = Note.new(message: nil)
28      note.valid?
29      expect(note.errors[:message]).to include("can't be blank")
30    end
31
32    describe "search message for a term" do
33      before do
34        @note1 = @project.notes.create(
35          message: "This is the first note.",
36          user: @user,
37        )
38        @note2 = @project.notes.create(
39          message: "This is the second note.",
40          user: @user,
41        )
42        @note3 = @project.notes.create(
43          message: "First, preheat the oven.",
44          user: @user,
45        )
46      end
47
48      context "when a match is found" do
49        it "returns notes that match the search term" do
50          expect(Note.search("first")).to include(@note1, @note3)
51        end
52      end
53
```

```
54        context "when no match is found" do
55          it "returns an empty collection" do
56            expect(Note.search("message")).to be_empty
57          end
58        end
59      end
60    end
```

You may notice one subtle difference in how we set up our test data. After moving
setup out of individual tests and into the before block, we need to assign each user to
an instance variable. Otherwise, we can't access them by variable name in the tests.

When we run the specs we'll see a nice outline (since we told RSpec to use the
documentation format, in chapter 2) like this:

```
Note
  is valid with a user, project, and message
  is invalid without a message
  search message for a term
    when a match is found
      returns notes that match the search term
    when no match is found
      returns an empty collection

Project
  does not allow duplicate project names per user
  allows two users to share a project name

User
  is valid with a first name, last name and email, and password
  is invalid without a first name
  is invalid without a last name
  is invalid with a duplicate email address
  returns a user's full name as a string

Finished in 0.22564 seconds (files took 0.32225 seconds to load)
11 examples, 0 failures
```

Some developers prefer to use method names for the descriptions of nested `describe` blocks. For example, I could have labeled `search for first name, last name, or email` as `#search`. I don't like doing this personally, as I believe the label should define the behavior of the code and not the name of the method. That said, I don't have a strong opinion about it.

How DRY is too DRY?

We've spent a lot of time in this chapter organizing specs into easy-to-follow blocks. This is an easy feature to abuse, though.

When setting up test conditions for your example, I think it's okay to bend the DRY principle in the interest of readability. If you find yourself scrolling up and down a large spec file in order to see what it is you're testing (or, later, loading too many external support files for your tests), consider duplicating your test data setup within smaller `describe` blocks—or even within examples themselves.

That said, well-named variables and methods can also go a long way—for example, in the spec above we used `@note1` and `@note1` as test notes, but in some cases, you may want to use variables like `@matching_note` or `@note_with_numbers_only`. It depends on what you're testing, but as a general rule, try to be expressive with your variable and method names!

We'll cover this topic in more depth in chapter 8.

Summary

This chapter focused on testing models, but we've covered a lot of other important techniques you'll want to use in other types of specs moving forward:

- **Use active, explicit expectations**: Use verbs to explain what an example's results should be. Only check for one result per example.
- **Test for what you expect *to* happen, and for what you expect *to not* happen**: Think about both paths when writing examples, and test accordingly.

- **Test for edge cases**: If you have a validation that requires a password be between four and ten characters in length, don't just test an eight-character password and call it good. A good set of tests would test at four and ten, as well as at three and eleven. (Of course, you might also take the opportunity to ask yourself why you'd allow such short passwords, or not allow longer ones. Testing is also a good opportunity to reflect on an application's requirements and code.)
- **Organize your specs for good readability**: Use `describe` and `context` to sort similar examples into an outline format, and `before` and `after` blocks to remove duplication. However, in the case of tests, readability is more important than DRY–if you find yourself having to scroll up and down your spec too much, it's okay to repeat yourself a bit.

With a solid collection of model specs incorporated into your app, you're well on your way to more trustworthy code.

Question

When should I use describe versus context? From RSpec's perspective, you can use `describe` all the time, if you'd like. Like many other aspects of RSpec, `context` exists to make your specs more readable. You could take advantage of this to match a condition, as I've done in this chapter, or some other state[12] in your application.

Exercise

Add more model tests to the sample application. I've only added tests to parts of our models' functionality. For example, the Project model's spec is lacking coverage of validations. Try adding them now. If you've got your own application configured to test with RSpec, try adding some model specs there, too.

---

[12]http://lmws.net/describe-vs-context-in-rspec

About Everyday Rails

Everyday Rails is a blog about using the Ruby on Rails web application framework to get stuff done as a web developer. It's about finding the best tools and techniques to get the most from Rails and help you get your apps to production. Everyday Rails can be found at http://everydayrails.com/

About the author

Aaron Sumner has developed web applications for more than 20 years. In that time he's gone from developing CGI with AppleScript (seriously) to Perl to PHP to Ruby and Rails. When off the clock and away from the text editor, Aaron enjoys photography, baseball (go Cards), college sports (Rock Chalk Jayhawk), outdoor cooking, woodworking, and bowling. He lives with his wife, Elise, along with five cats and a dog in rural Kansas.

Aaron's personal blog is at https://www.aaronsumner.com/. *Everyday Rails Testing with RSpec* is his first book.

Colophon

The cover image of a practical, reliable, red pickup truck[13] is by iStockphoto contributor Habman_18[14]. I spent a lot of time reviewing photos for the cover–too much time, probably–but picked this one because it represents my approach to Rails testing–not flashy, and maybe not always the fastest way to get there, but solid and dependable. And it's red, like Ruby. Maybe it should have been green, like a passing spec? Hmm.

---

[13]http://www.istockphoto.com/stock-photo-16071171-old-truck-in-early-morning-light.php?st=1e7555f
[14]http://www.istockphoto.com/user_view.php?id=4151137