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 2014-12-20

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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

Preface to this edition

Here it is, the RSpec 3 edition of *Everyday Rails Testing with RSpec*! A lot has changed, and I hope you'll find it worth the wait.

As with previous updates, I've rebuilt the sample application to use current versions of Rails, RSpec, and the other gems used throughout the book. I also expanded some sections–most notably, chapter 10, *Testing the Rest*. I also updated as needed to take advantage of new features in RSpec 3 and Rails 4.1. 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[1] for this release and I'll address them promptly.

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.

[1] https://github.com/everydayrails/rails-4-1-rspec-3-0/issues

1. Introduction

Ruby on Rails and automated testing go hand in hand. Rails ships with a built-in test framework; if it's not to your liking you can replace it with one of your liking. As I write this, Ruby Toolbox lists 17 projects under the *Unit Test Frameworks* category[2] alone. So, yeah, testing's pretty important in Rails. Yet, many people developing in Rails are either not testing their projects at all, or at best only adding a few token specs on model validations.

In my opinion, there are several reasons for this. Perhaps working with Ruby or 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. Historically, I haven't 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, but usually 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

[2] https://www.ruby-toolbox.com/categories/testing_frameworks

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

complex applications in a fraction of the time it took me in my previous framework-less efforts. I was hooked.

That said, early Rails books and tutorials focused more on 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 now 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 boat to the one I was in–may find themselves without a consistent testing strategy.

My 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.

Why RSpec?

For the most part, I have nothing against the other test frameworks out there. If I'm writing a standalone Ruby library, I usually rely on MiniTest. For whatever reason, though, RSpec is the one that's stuck with me when it comes to testing my Rails applications.

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.

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* (look for the Rails 4-specific version online[4]), Daniel Kehoe's *Learn*

[4] http://ruby.railstutorial.org

Ruby on Rails, or Sam Ruby's *Agile Web Development with Rails 4*, before digging into *Everyday Rails Testing with RSpec*–this book assumes you've got some basic Rails skills under your belt. In other words, this book 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–we're going to be installing RSpec and a few extras to make the testing process as easy as possible to comprehend and manage.

If you've been developing in Rails for a little while, and maybe even have an application or two in production–but testing is still a foreign concept–this book is for you! I was in your shoes 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

- Model-View-Controller application architecture, as used in Rails
- Bundler for gem dependency management
- How to run Rake tasks
- Basic command line tools
- Enough Git to switch between branches of a repository

On the more advanced end, if you're familiar with using Test::Unit, MiniTest, or even RSpec itself, and already have a workflow in place that (a) you're comfortable with and (b) provides adequate coverage, you may be able to fine-tune some of your approach to testing your applications. But to be honest, at this point you're probably on board with automated testing and don't need this extra nudge. This is not a book on testing theory; it also won't dig too deeply into performance issues. Other books may be of more use to you in the long run.

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

Discussing the *right* way to test your Rails application can invoke major shouting matches amongst programmers–not quite as bad as, say, the Vim versus Emacs debate, 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," has sparked a fresh round of debates on the topic.

So, yes, there is a right way to do testing–but if you ask me, there are degrees of *right* when it comes to testing.

At the risk of starting additional riots among the Ruby test-driven/behavior-driven development communities, my approach focuses on the following foundation:

- Tests should be reliable.
- Tests should be easy to write.
- Tests should be easy to understand.

If you mind these three factors in your approach, you'll go a long way toward having a sound test suite for your application–not to mention becoming an honest-to-goodness practitioner of Test-Driven Development. Whatever that means these days.

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 tests–and reliable, understandable tests, even if they're not quite as optimized as they could be, are a great way to start. It's 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 *Everyday Rails Testing with RSpec* I'll walk you through taking a basic Rails 4.1 application from completely untested to respectably tested with RSpec 3.1. 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, along with a few extra, useful testing tools.
- In chapter 3, *Model Specs*, we'll tackle testing our application's models through reliable unit testing.
- Chapter 4, *Generating Test Data with Factories*, covers factories, making test data generation straightforward.
- We'll take an initial look at testing controllers in chapter 5, *Basic Controller Specs*.
- Chapter 6, *Advanced Controller Specs*, is about using controller specs to make sure your authentication and authorization layers are doing their jobs–that is, keeping your app's data safe.
- Chapter 7, *Controller Spec Cleanup*, is our first round of spec refactoring, reducing redundancy without removing readability.
- In chapter 8, *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 9, *Speeding up specs*, we'll go over some techniques for refactoring and running your tests with performance in mind.
- 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 APIs.
- 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 question-and-answer section, 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

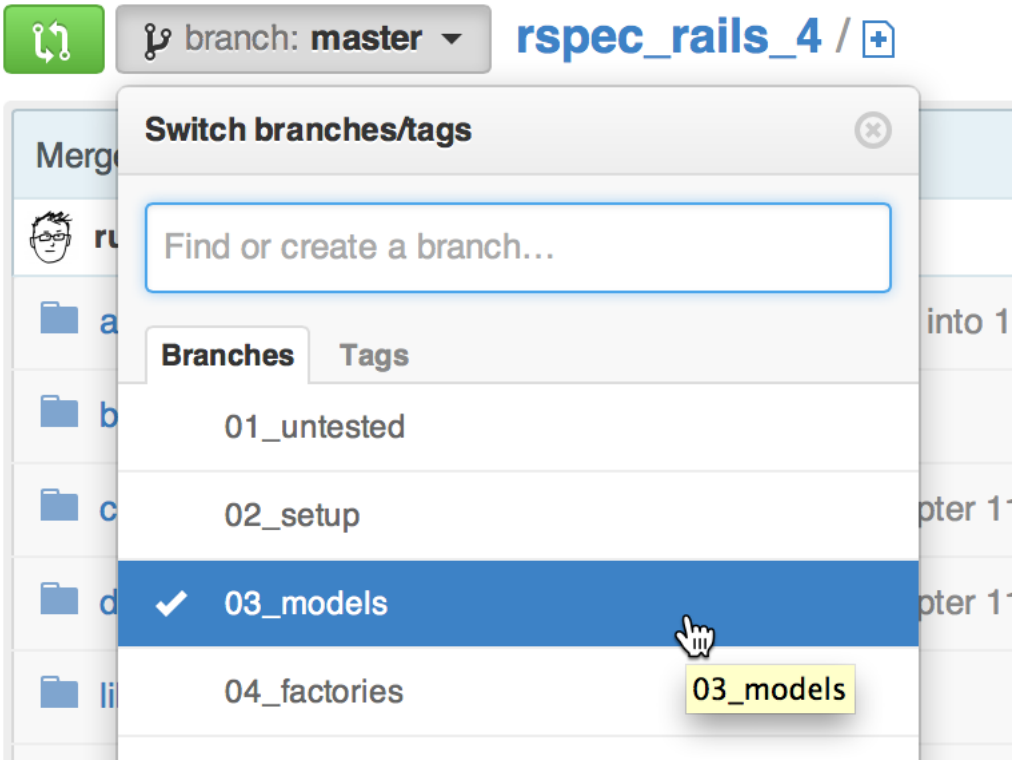
Speaking of the sample code, you can find a completely tested application on GitHub.

Get the source!

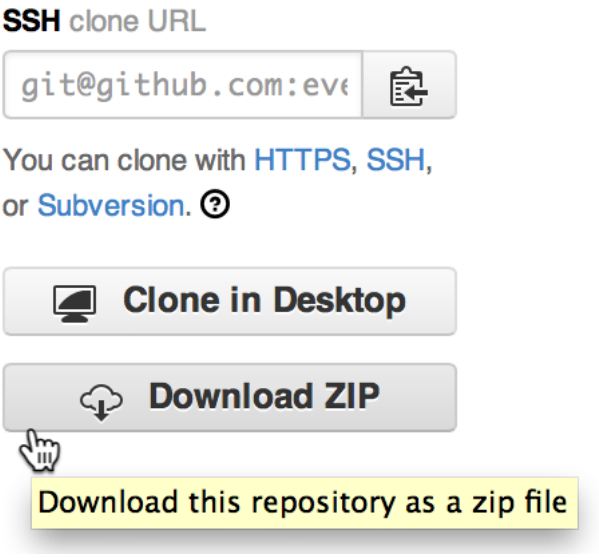
https://github.com/everydayrails/rails-4-1-rspec-3-0

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, but I'll also tell you which branch to check out at the start of that chapter.

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:



Git Immersion[5] is an excellent, hands-on way to learn the basics of Git on the command line. So is Try Git[6]. For a handy refresher of the basics, check out Git Reference[7].

Code conventions

I'm using the following setup for this application:

- **Rails 4.1**: The latest version of Rails is the big focus of this book; however, as far as I know 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 2.1**: I don't think you'll see any major differences if you're using 1.9 or 2.0. At this point I don't recommend trying to progress through the book if you're still using Ruby 1.8.

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

- **RSpec 3.1**: RSpec 3.0 was released in spring, 2014. RSpec 3.1 appeared a few months later and is by and large 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 any of the above, 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; rather, 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[8]–*Everyday Rails Testing 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 the interwebs or an ebook is a better way to learn.

Discussion and errata

Nobody's perfect, especially not me. I've put a lot of time and effort into making sure *Everyday Rails Testing 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/rails-4-1-rspec-3-0/issues

About the sample application

Our sample application is an admittedly simple, admittedly ugly little contacts manager, perhaps part of a corporate website. The application lists names, email addresses, and phone numbers to anyone who comes across the site, and also provides a simple, first-letter search function. Users must log in to add new contacts or make

[8] http://learncodethehardway.org/

changes to existing ones. Finally, users must have an administrator ability to add new users to the system.

Up to this point, though, I've been intentionally lazy and only used Rails' default generators to create the entire application (see the *01_untested* branch of the sample code). This means I have a *test* directory full of untouched test files and fixtures. I could run `rake test` at this point, and perhaps some of these tests would even pass. But since this is a book about RSpec, a better solution will be to dump this folder, set up Rails to use RSpec instead, and build out a more respectable 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 and to start generating the appropriate specs (and a few other useful files) whenever we employ a Rails generator to add code to the application.

Let's get started!

3. Model specs

We've got all the tools we need for building a solid, reliable test suite–now it's time to put them to work. 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 a model spec for an existing model–in our case, the actual *Contact* model.
- Then, we'll write passing tests for a model's validations, class, and instance methods, and organize our spec in the process.

We'll create our first spec files for existing models by hand. If and when we add new models to the application (OK, when we do in chapter 11), the handy RSpec generators we configured in chapter 2 will generate placeholder files for us.

Check out the *03_models* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 03_models origin/03_models
```

If you'd like to follow along, start with the previous chapter's branch:

```
git checkout -b 02_setup origin/02_setup
```

See chapter 1 for additional details.

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 is key–a solid foundation is the first step toward a reliable overall code base.

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

- The model's `create` method, when passed valid attributes, 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 main *Contact* model's requirements:

```
describe Contact do
  it "is valid with a firstname, lastname and email"
  it "is invalid without a firstname"
  it "is invalid without a lastname"
  it "is invalid without an email address"
  it "is invalid with a duplicate email address"
  it "returns a contact's full name as a string"
end
```

We'll expand this outline in a few minutes, but this gives us quite a bit 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 `Contact` 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 `firstname`, `lastname`, and `email` validations separately. This way, if an example fails, I know it's because of that *specific* validation, and 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: *Contact is invalid without a firstname*, *Contact is invalid without a lastname*, *Contact returns a contact's full name as a string*. Readability is important!

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

Creating a model spec

First, we'll open up the `spec` directory and, if necessary, create a subdirectory named `models`. Inside that subdirectory let's create a file named `contact_spec.rb` and add the following:

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4    it "is valid with a firstname, lastname and email"
5    it "is invalid without a firstname"
6    it "is invalid without a lastname"
7    it "is invalid without an email address"
8    it "is invalid with a duplicate email address"
9    it "returns a contact's full name as a string"
10 end
```

Notice the `require 'rails_helper'` at the top, and get used to typing it–all of your specs will include this line moving forward. This is a new addition to RSpec 3–previous version required `spec_helper`, but the Rails-specific details have been extracted to make the main helper significantly lighter. We'll touch on this a little further in chapter 9.

Location, location, location

The name and location for your spec file is important! RSpec's file structure mirrors that of the `app` directory, as do the files within it. In the case of model specs, `contact_spec.rb` should correspond to `contact.rb`. This becomes more important later when we start automating tests to run as soon as a spec's corresponding application file is updated, and vice versa.

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` or just `rspec` on the command line) the output would be similar to the following:

```
Contact
  is valid with a firstname, lastname and email
    (PENDING: Not yet implemented)
  is invalid without a firstname
    (PENDING: Not yet implemented)
  is invalid without a lastname
    (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 contact's full name as a string
    (PENDING: Not yet implemented)

Pending:
  Contact is valid with a firstname, lastname
    and email
    # Not yet implemented
    # ./spec/models/contact_spec.rb:4
  Contact is invalid without a firstname
    # Not yet implemented
    # ./spec/models/contact_spec.rb:5
  Contact is invalid without a lastname
    # Not yet implemented
    # ./spec/models/contact_spec.rb:6
  Contact is invalid without an email address
    # Not yet implemented
    # ./spec/models/contact_spec.rb:7
  Contact is invalid with a duplicate email address
    # Not yet implemented
    # ./spec/models/contact_spec.rb:8
  Contact returns a contact's full name as a string
    # Not yet implemented
    # ./spec/models/contact_spec.rb:9

Finished in 0.00105 seconds (files took 2.42 seconds to load)
6 examples, 0 failures, 6 pending
```

Great! Six pending specs–let's write them and make them pass, starting with the first example.

As we add additional models to the contacts manager, assuming we use Rails' `model` or `scaffold` generator to do so, the model spec file will be added automatically. If it doesn't go back and configure your application's generators now, or make sure you've properly installed the *rspec-rails* gem, as shown in chapter 2. You'll still need to fill in the details, though.

The new RSpec syntax

In June, 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[9]. 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 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:

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

```
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, there's still a good chance you'll find information using the old `should` syntax. This syntax still technically works in RSpec 3, 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 `Contact` model:

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4    it "is valid with a firstname, lastname and email" do
5      contact = Contact.new(
6        firstname: 'Aaron',
7        lastname: 'Sumner',
8        email: 'tester@example.com')
9      expect(contact).to be_valid
10   end
11
12   # remaining examples to come
13 end
```

This simple example uses RSpec's `be_valid` matcher 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 `Contact` called `contact`), then pass that to `expect` to compare to the matcher.

Now, if we run RSpec from the command line again (via `bin/rspec` or `bundle exec rspec`, depending on whether you installed the `rspec` binstub in the previous chapter) we see one passing example! We're on our way. Now let's get into testing more of our code.

Testing validations

Validations are a good way to break into automated testing. These tests can usually be written in just a line or two of code, especially when we leverage the convenience of factories (next chapter). Let's look at some detail to our `firstname` validation spec:

spec/models/contact_spec.rb

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

This time, we *expect* that the new contact (with a *firstname* explicitly set to `nil`) will not be valid, thus returning the shown error message on the contact's *firstname* 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.

To prove that we're not getting false positives, let's flip that expectation by changing `to` to `not_to`:

spec/models/contact_spec.rb

```
1  it "is invalid without a firstname" do
2    contact = Contact.new(firstname: nil)
3    contact.valid?
4    expect(contact.errors[:firstname]).not_to include("can't be blank")
5  end
```

And sure enough, RSpec reports a failure:

```
Failures:

  1) Contact is invalid without a firstname
     Failure/Error: expect(contact.errors[:firstname]).not_to
         include("can't be blank")
       expected ["can't be blank"] not to include "can't be blank"
     # ./spec/models/contact_spec.rb:15:in `block (2 levels) in
         <top (required)>'
```

RSpec provides `not_to` and `to_not` for these types of expectations. They're interchangeable. I use `not_to` in the book.

This is an easy way to verify your tests are working correctly, especially as you progress from testing simple validations to more complex logic. Just remember to flip that `not_to` back to `to` before continuing.

If you've used an earlier version of RSpec, you may be used to using the `have` matcher and `errors_on` helper method to check for validation errors. These have been removed from RSpec 3's core. You can still use the `have` matcher by including `rspec-collection_matchers` in your Gemfile's `:test` group.

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

spec/models/contact_spec.rb

```
1  it "is invalid without a lastname" do
2    contact = Contact.new(lastname: nil)
3    contact.valid?
4    expect(contact.errors[:lastname]).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.

Testing that email addresses must be unique is fairly simple as well:

spec/models/contact_spec.rb

```
it "is invalid with a duplicate email address" do
  Contact.create(
    firstname: 'Joe', lastname: 'Tester',
    email: 'tester@example.com'
  )
  contact = Contact.new(
    firstname: 'Jane', lastname: 'Tester',
    email: 'tester@example.com'
  )
  contact.valid?
  expect(contact.errors[:email]).to include("has already been taken")
end
```

Notice a subtle difference here: In this case, we persisted a contact (calling `create` on `Contact` instead of `new`) to test against, then instantiated a second contact as the subject of the actual test. This, of course, requires that the first, persisted contact is valid (with both a first and last name) and has an email address assigned to it. In future chapters we'll look at utilities to streamline this process.

Now let's test a more complex validation. Say we want to make sure we don't duplicate a phone number for a user–their home, office, and mobile phones should all be unique within the scope of that user. How might you test that?

Switching to the *Phone* model spec, we have the following example:

spec/models/phone_spec.rb

```
require 'rails_helper'

describe Phone do
  it "does not allow duplicate phone numbers per contact" do
    contact = Contact.create(
      firstname: 'Joe',
      lastname: 'Tester',
      email: 'joetester@example.com'
    )
    contact.phones.create(
      phone_type: 'home',
      phone: '785-555-1234'
    )
    mobile_phone = contact.phones.build(
      phone_type: 'mobile',
      phone: '785-555-1234'
    )

    mobile_phone.valid?
    expect(mobile_phone.errors[:phone]).to include('has already been taken')
  end

  it "allows two contacts to share a phone number" do
    contact = Contact.create(
      firstname: 'Joe',
      lastname: 'Tester',
      email: 'joetester@example.com'
    )
    contact.phones.create(
      phone_type: 'home',
      phone: '785-555-1234'
    )
    other_contact = Contact.new
    other_phone = other_contact.phones.build(
      phone_type: 'home',
      phone: '785-555-1234'
```

```
37      )
38
39      expect(other_phone).to be_valid
40    end
41  end
```

This time, since the `Contact` and `Phone` 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 contact to which both phones are assigned. In the second, the same phone number is assigned to two unique contacts. Note that, in both examples, we have to *create* the contact, or persist it in the database, in order to assign it to the phones we're testing.

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

app/models/phone.rb

```
validates :phone, uniqueness: { scope: :contact_id }
```

These specs will pass without issue.

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.

Testing instance methods

It would be convenient to only have to refer to `@contact.name` to render our contacts' full names instead of concatenating the first and last names into a new string every time, so we've got this method in the `Contact` class:

app/models/contact.rb

```
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/contact_spec.rb

```
it "returns a contact's full name as a string" do
  contact = Contact.new(firstname: 'John', lastname: 'Doe',
    email: 'johndoe@example.com')
  expect(contact.name).to eq 'John Doe'
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

Now let's test the *Contact* model's ability to return a list of contacts whose names begin with a given letter. For example, if I click *S* then I should get *Smith, Sumner*, and so on, but not *Jones*. There are a number of ways I could implement this–for demonstration purposes I'll show one.

The model implements this functionality in the following simple method:

app/models/contact.rb

```
def self.by_letter(letter)
  where("lastname LIKE ?", "#{letter}%").order(:lastname)
end
```

To test this, let's add the following to our *Contact* spec:

spec/models/contact_spec.rb

```
require 'rails_helper'

describe Contact do

  # earlier validation examples omitted ...

  it "returns a sorted array of results that match" do
    smith = Contact.create(
      firstname: 'John',
      lastname: 'Smith',
      email: 'jsmith@example.com'
    )
    jones = Contact.create(
      firstname: 'Tim',
      lastname: 'Jones',
      email: 'tjones@example.com'
    )
    johnson = Contact.create(
      firstname: 'John',
      lastname: 'Johnson',
      email: 'jjohnson@example.com'
    )
    expect(Contact.by_letter("J")).to eq [johnson, jones]
  end
end
```

Note we're testing both the results of the query and the sort order; `jones` will be retrieved from the database first but since we're sorting by last name then `johnson` should be stored first in the query results.

Testing for failures

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

spec/models/contact_spec.rb

```
require 'rails_helper'

describe Contact do

  # validation examples ...

  it "omits results that do not match" do
    smith = Contact.create(
      firstname: 'John',
      lastname: 'Smith',
      email: 'jsmith@example.com'
    )
    jones = Contact.create(
      firstname: 'Tim',
      lastname: 'Jones',
      email: 'tjones@example.com'
    )
    johnson = Contact.create(
      firstname: 'John',
      lastname: 'Johnson',
      email: 'jjohnson@example.com'
    )
    expect(Contact.by_letter("J")).not_to include smith
  end
end
```

This spec uses RSpec's `include` matcher to determine if the array returned by `Contact.by_letter("J")`–and it passes! We're testing not just for ideal results–the user selects a letter with results–but also for letters with no results.

More about matchers

We've already seen three matchers in action. 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[10]. And in chapter 7, we'll take a look at creating custom matchers of our own.

DRYer specs with describe, context, before and after

If you're following along with the sample code, you've no doubt spotted a discrepancy there with what we've covered here. In that code, I'm using yet another RSpec feature, the `before` hook, to help simplify the spec's code and reduce typing. Indeed, the spec samples have some redundancy: We create the same three 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 RSpec tricks to clean things up.

The first thing I'm going to do is create a `describe` block *within* my `describe Contact` block to focus on the filter feature. The general outline will look like this:

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

spec/models/contact_spec.rb

```
require 'rails_helper'

describe Contact do

  # validation examples ...

  describe "filter last name by letter" do
    # filtering examples ...
  end
end
```

Let's break things down even further by including a couple of `context` blocks–one for matching letters, one for non-matching:

spec/models/contact_spec.rb

```
require 'rails_helper'

describe Contact do

  # validation examples ...

  describe "filter last name by letter" do
    context "matching letters" do
      # matching examples ...
    end

    context "non-matching letters" do
      # non-matching examples ...
    end
  end
end
```

While `describe` and `context` are technically interchangeable, I prefer to use them like this–specifically, `describe` outlines general functionality of my class; `context` outlines a specific state. In my case, I have a state of a letter with matching results selected, and a state with a non-matching letter 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:

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4
5    # validation examples ...
6
7    describe "filter last name by letter" do
8      before :each do
9        @smith = Contact.create(
10         firstname: 'John',
11         lastname: 'Smith',
12         email: 'jsmith@example.com'
13       )
14       @jones = Contact.create(
15         firstname: 'Tim',
16         lastname: 'Jones',
17         email: 'tjones@example.com'
18       )
19       @johnson = Contact.create(
20         firstname: 'John',
21         lastname: 'Johnson',
22         email: 'jjohnson@example.com'
23       )
24     end
25
26     context "matching letters" do
```

```
27        # matching examples ...
28      end
29
30      context "non-matching letters" do
31        # non-matching examples ...
32      end
33    end
34  end
```

RSpec's `before` hooks are vital to cleaning up nasty redundancy from your specs. As you might guess, the code contained within the `before` block is run before *each* example within the `describe` block–but not outside of that block. Since we've indicated that the hook should be run before *each* example within the block, RSpec will create them for each example individually. In this example, my `before` hook will *only* be called within the `describe "filter last name by letter"` block–in other words, my original validation specs will not have access to `@smith`, `@jones`, and `@johnson`.

`:each` is the default behavior of `before`, and many Rubyists use the shorter `before do` to create `before` blocks. I prefer the explicitness of `before :each do` and will use it throughout the book.

Speaking of my three test contacts, note that since they are no longer being created within each example, we have to assign them to instance variables, so they're accessible outside of the `before` block, within our actual examples.

If a spec requires some sort of post-example teardown–disconnecting from an external service, say–we can also use an `after` hook to clean up after the examples. Since RSpec handles cleaning up the database by default, I rarely use `after`. `before`, though, is indispensable.

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

spec/models/contact_spec.rb

```
require 'rails_helper'

describe Contact do
  it "is valid with a firstname, lastname and email" do
    contact = Contact.new(
      firstname: 'Aaron',
      lastname: 'Sumner',
      email: 'tester@example.com')
    expect(contact).to be_valid
  end

  it "is invalid without a firstname" do
    contact = Contact.new(firstname: nil)
    contact.valid?
    expect(contact.errors[:firstname]).to include("can't be blank")
  end

  it "is invalid without a lastname" do
    contact = Contact.new(lastname: nil)
    contact.valid?
    expect(contact.errors[:lastname]).to include("can't be blank")
  end

  it "is invalid without an email address" do
    contact = Contact.new(email: nil)
    contact.valid?
    expect(contact.errors[:email]).to include("can't be blank")
  end

  it "is invalid with a duplicate email address" do
    Contact.create(
      firstname: 'Joe', lastname: 'Tester',
      email: 'tester@example.com'
    )
    contact = Contact.new(
      firstname: 'Jane', lastname: 'Tester',
```

```
37        email: 'tester@example.com'
38      )
39      contact.valid?
40      expect(contact.errors[:email]).to include("has already been taken")
41    end
42
43    it "returns a contact's full name as a string" do
44      contact = Contact.new(
45        firstname: 'John',
46        lastname: 'Doe',
47        email: 'johndoe@example.com'
48      )
49      expect(contact.name).to eq 'John Doe'
50    end
51
52    describe "filter last name by letter" do
53      before :each do
54        @smith = Contact.create(
55          firstname: 'John',
56          lastname: 'Smith',
57          email: 'jsmith@example.com'
58        )
59        @jones = Contact.create(
60          firstname: 'Tim',
61          lastname: 'Jones',
62          email: 'tjones@example.com'
63        )
64        @johnson = Contact.create(
65          firstname: 'John',
66          lastname: 'Johnson',
67          email: 'jjohnson@example.com'
68        )
69      end
70
71      context "with matching letters" do
72        it "returns a sorted array of results that match" do
73          expect(Contact.by_letter("J")).to eq [@johnson, @jones]
74        end
```

```
75      end
76
77      context "with non-matching letters" do
78        it "omits results that do not match" do
79          expect(Contact.by_letter("J")).not_to include @smith
80        end
81      end
82    end
83  end
```

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:

```
Contact
  is valid with a firstname, lastname and email
  is invalid without a firstname
  is invalid without a lastname
  is invalid without an email address
  is invalid with a duplicate email address
  returns a contact's full name as a string
  filter last name by letter
    with matching letters
      returns a sorted array of results that match
    with non-matching letters
      omits results that do not match

Phone
  does not allow duplicate phone numbers per contact
  allows two contacts to share a phone number

Finished in 0.51654 seconds (files took 2.24 seconds to load)
10 examples, 0 failures
```

Some developers prefer to use method names for the descriptions of nested `describe` blocks. For example, I could have labeled `filter last name by letter` as `#by_letter`. 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. Like I said, `before` blocks are key to making this happen–but they're also easy to abuse.

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 go a long way–for example, in the spec above we used `@jones` and `@johnson` as test contacts. These are much easier to follow than `@user1` and `@user2` would have been, as part of these test objects' value to the tests was to make sure our first-letter search functionality was working as intended. Even better might be variables like `@admin_user` and `@guest_user`, when we get into testing users with specific roles in chapter 6. Be expressive with your variable names!

Summary

This chapter focused on how I test 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 trumps 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. In the next chapter we'll apply and expand upon the techniques covered here to application controllers.

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[11] in your application.

Exercises

So far we've assumed our specs aren't returning false positives–they've all gone from pending to passing without failing somewhere in the middle. Verify specs by doing the following:

- **Comment out the application code you're testing**. For example, in our example that validates the presence of a contact's first name, we could comment out `validates :firstname, presence: true`, run the specs, and watch `it "is invalid without a firstname"` fail. Uncomment it to see the spec pass again.
- **Edit the parameters passed to the `create` method within the expectation**. This time, edit `it "is invalid without a firstname"` and give `:firstname` a non-`nil` value. The spec should fail; replace it with `nil` to see it pass again.

[11] 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 gems 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 is a Ruby developer in the heart of Django country. He's developed web applications since the mid-1990s. 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 basketball (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 http://www.aaronsumner.com/. *Everyday Rails Testing with RSpec* is his first book.

Colophon

The cover image of a practical, reliable, red pickup truck[12] is by iStockphoto contributor Habman_18[13]. 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.

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

[13] http://www.istockphoto.com/user_view.php?id=4151137