everyday Rails

Testing with RSpec

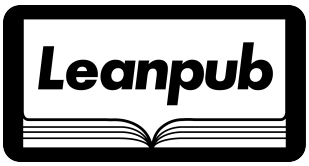
A practical approach to test-driven development

Aaron Sumner

Everyday Rails Testing with RSpec

A practical approach to test-driven development

This version was published on 2012-06-11



This is a Leanpub book, for sale at:

http://leanpub.com/everydayrailsrspec

Leanpub helps authors to self-publish in-progress ebooks. We call this idea Lean Publishing. To learn more about Lean Publishing, go to: http://leanpub.com/manifesto

To learn more about Leanpub, go to: http://leanpub.com

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/#everydayrailsrspec

Contents

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 16 projects under the *Unit Test Frameworks* category[1] 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; 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 clicking links in the browser is too hard to break.

I've been there. I don't consider myself an engineer in the traditional sense, yet I have problems to solve and typically find solutions 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 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[2] 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.

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. Newer works on Rails have addressed this shortcoming, and now demonstrate how to test applications throughout, and 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 adapt to make work consistently for you, too.

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

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

Why RSpec?

Nothing against the other test frameworks out there, but for whatever reason RSpec is the one that's stuck with me. 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 testing in the likes of Michael Hartl's *Rails 3 Tutorial* or Sam Ruby's *Agile Web Development with Rails (4th Edition)* 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 a few extras to make the testing process easier 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

- MVC architecture, as used in Rails
- Bundler
- How to run rake tasks
- Basic command line techniques

On the more advanced end, if you're familiar with using Test::Unit, MiniTest, or even RSpec itself, and have a workflow that (a) you're comfortable with and (b) provides adequate coverage already in place, 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, like David Chelimsky's *The RSpec Book* or Noel Rappin's *Rails Test Prescriptions*, 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 holy wars–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. 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 riots among the Ruby TDD and BDD 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.

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 (and 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 3.2 application from completely untested to respectably tested with RSpec. 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, *More on Factories*, covers factories with a bit more depth, 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, *Integration Testing with Request Specs*, we'll move on to integration testing with request 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, and time-specific functionality.
- I'll talk about what it means to practice 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 guide 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–we won't be building an application together in this book, just exploring code patterns and techniques.

You can download the source from the Everyday Rails website at

http://everydayrails.com/rspecbook/code.zip.

I've created a separate version of the application for each stage along this process. The number in each project's name (e.g., *02_setup*, *03_models*, etc.) corresponds to the chapter covering the changes applied to that version. While you should be able to spin any version up with relative ease (dependencies are minimal) and type along, I strongly recommend applying what you learn from this book in your own applications instead.

Just so you know, I use *we* a lot throughout the book as opposed to *I* or *you* or the third person. I've written the book as if you and I are sitting at a display together, working through the process. Think of *we* as a conversation between two colleagues–I'm definitely not the type to refer to myself as *we*.

Code conventions

I'm using the following setup for this application:

- **Rails 3.2:** 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.
- **Ruby 1.9:** Again, the basic techniques will work; you'll just need to be mindful that I'll be using the Ruby 1.9 hash syntax and adjust accordingly if your application uses Ruby 1.8. (In other words, switch out the `key: value` syntax with `:key => value`.)
- **RSpec 2.10:** I began writing this book while version 2.8 was current, so anything you see here should apply to versions of RSpec at least that far back.

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[3]–*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.

[3] http://learncodethehardway.org/

About the 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 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 *code/01_untested*). 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, tell 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.
- Next, we'll simplify the process of creating and maintaining test data with *factories*.
- Finally, 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 and factories for existing models by hand. If and when we add new models to the application, the handy RSpec generators we configured in chapter 2 will generate placeholder files for us.

Follow along with the version of the sample application located in *code/03_models*.

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. (An object-oriented application without objects isn't very useful, after all.) 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 default factory should generate a valid object (more on factories in just a moment).
- 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
  it "has a valid factory"
  it "is invalid without a firstname"
  it "is invalid without a lastname"
  it "returns a contact's full name as a string"
```

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 three best practices:

- **Each example (a line beginning with `it`) only expects one thing**. Notice that I'm testing the `firstname` and `lastname` 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**. *Should* is redundant here, and clutters RSpec's output. Omitting it makes specs' output easier to read.

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

require 'spec_helper'

describe Contact do
  it "has a valid factory"
  it "is invalid without a firstname"
  it "is invalid without a lastname"
  it "returns a contact's full name as a string"
end
```

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 things.

We'll fill in the details in a moment, but if we ran the specs right now from the command line (using `bundle exec rspec`) the output would be similar to the following:

```
Contact
  has a valid factory (PENDING: Not yet implemented)
  is invalid without a firstname (PENDING: Not yet implemented)
  is invalid without a lastname (PENDING: Not yet implemented)
  returns a contact's full name as a string (PENDING: Not yet implemented)

Pending:
  Contact has a valid factory
    # 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 returns a contact's full name as a string
    # Not yet implemented
    # ./spec/models/contact_spec.rb:7

Finished in 0.00045 seconds
4 examples, 0 failures, 4 pending
```

Great! Four pending specs–let's write them and make them pass.

As we add additional models to the contacts manager, assuming we use Rails' `model` generator to do so, the model spec file (along with an associated factory) 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* and *factory_girl_rails* gems, as shown in chapter 2.)

Generating test data with factories

I won't spend a lot of time bad-mouthing fixtures–frankly, it's already been done by plenty of people smarter than me in the Rails testing community. Long story short, there are two issues presented by fixtures I'd like to avoid: First, fixture data can be brittle and easily broken (meaning you spend about as much time maintaining your test data as you do your tests and actual code); and second, Rails bypasses Active Record when it loads fixture data into your test database. What does that mean? It means that important things like your models' validations are ignored. This is bad!

Enter **factories**: Simple, flexible, building blocks for test data. If I had to point to a single component that helped me see the light toward testing more than anything else, it would be Factory Girl[4], an easy-to-use and easy-to-rely-on gem for creating test data without the brittleness of fixtures. Since we've got Factory Girl installed courtesy of the *factory_girl_rails* gem we installed earlier, we've got full access to factories in our app. Let's put them to work!

Back in the *spec* directory, add another subdirectory named *factories*; within it, add the file *contacts.rb* with the following content:

```
1  #spec/factories/contacts.rb
2
3  FactoryGirl.define do
4    factory :contact do
5      firstname "John"
6      lastname "Doe"
7      sequence(:email) { |n| "johndoe#{n}@example.com"}
8    end
9  end
```

This chunk of code gives us a *factory* we can use throughout our specs. Essentially, whenever we create test data via `FactoryGirl.create(:contact)`, that contact's name will be *John Doe*. His email address? We're using a handy feature provided by Factory Girl, called **sequences**. As you might have guessed from reading the code, a sequence will automatically increment `n` inside the block, yielding `johndoe1@example.com`, `johndoe2@example.com`, and so on as the factory is used to generate new contacts. Sequences are essential for any model that has a `uniqueness` validation (in the next chapter, we'll look at a nice alternative to generating things like email addresses and names, called Faker).

Filenames for factories aren't as particular as those for specs. In fact, if you wanted to you could include all of your factories in a single file. However, the Factory Girl generator stores them in *spec/factories* as convention, with a filename that's the plural

[4] https://github.com/thoughtbot/factory_girl

of the model it corresponds to (so, *spec/factories/contacts.rb* for the Contact model). I tend to just stick with that approach, too.

With a solid factory in place, let's return to the *contact_spec.rb* file we set up a few minutes ago and locate the first example (`it "has a valid factory"`). We're going to write our first spec–essentially testing the factory we just created. In the sample code look at the following spec:

```
1  # spec/models/contact_spec.rb
2
3  require 'spec_helper'
4
5  describe Contact do
6    it "has a valid factory" do
7      FactoryGirl.create(:contact).should be_valid
8    end
9    it "is invalid without a firstname"
10   it "is invalid without a lastname"
11   it "returns a contact's full name as a string"
12 end
```

This single-line spec uses RSpec's `be_valid` matcher to verify that our new factory does indeed return a valid contact.

One last step: Even though we've got a test database, it doesn't have a schema. To make it match the development schema, run the following rake task:

```
1  rake db:test:clone
```

This "clones" the database structure as used in development to production. However, the task doesn't copy any *data*–any data setup that a given test requires will be up to you, as we'll see throughout this book.

Any time you make a change to your development database, you'll need to mirror that change in your test database with `rake db:test:clone`. If you run rspec and get an error about an unknown database, it's probably because you haven't cloned yet.

Now, if we run RSpec from the command line again we see one passing example! We're on our way. Now let's get into testing the code we actually wrote.

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. Let's look at some detail to our `firstname` validation spec:

```
# spec/models/contact_spec.rb

it "is invalid without a firstname" do
  FactoryGirl.build(:contact, firstname: nil).should_not be_valid
end
```

Note what we're doing with Factory Girl here: First, instead of the `FactoryGirl.create()` approach, we're using `FactoryGirl.build()`. Can you guess the difference? `FactoryGirl.create()` builds the model and saves it, while `FactoryGirl.build()` instantiates a new model, but doesn't save it. If we used `FactoryGirl.create()` in this example it would break before we could even run the test, due to the currently failing validation.

Second, we use the `Contact` factory's defaults for every attribute except `:firstname`, and for that we pass `nil` to give it no value. In other words, instead of the default name of *John Doe* our `Contact` factory would normally give us, it returns *Doe*. This is an incredibly convenient feature, especially when testing at the model level, so let me state it again: You can override any of a factory's default values by passing in a different one in your spec. You'll use it a lot in your tests–starting with models, but more in other tests, too.

When we run RSpec again; we should be up to two passing specs. Now we can use the same approach to test the `:lastname` validation.

```
# spec/models/contact_spec.rb

it "is invalid without a lastname" do
  FactoryGirl.build(:contact, lastname: nil).should_not be_valid
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. If you think about what validations your model should have *while* writing tests (ideally, and eventually, in a Test-Driven Development pattern), you are more likely to remember to include them.

Testing that email addresses must be unique is fairly simple as well–again, we'll override the factory's defaults with our own values:

```
# spec/models/contact_spec.rb

it "is invalid with a duplicate email address" do
  FactoryGirl.create(:contact, email: "aaron@everydayrails.com")
  FactoryGirl.build(:contact,
    email: "aaron@everydayrails.com").should_not be_valid
end
```

In this case, we persisted a contact to test against, then created a second contact as the subject of the actual test.

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 to them. How might you test that?

In the *Phone* model spec, we have the following example:

```
# spec/models/phone_spec.rb

it "does not allow duplicate phone numbers per contact" do
  contact = FactoryGirl.create(:contact)
  FactoryGirl.create(:phone,
    contact: contact,
    phone_type: "home",
    phone: "785-555-1234")
  FactoryGirl.build(:phone,
    contact: contact,
    phone_type: "mobile",
    phone: "785-555-1234").should_not be_valid
end
```

And the following factory:

```
#spec/factories/phones.rb

FactoryGirl.define do
  factory :phone do
    association :contact
    phone { '123-555-1234' }
    phone_type 'home'
  end
end
```

And it should pass. Let's take a quick look at that last factory, though–it's got a new component, `association`, and assigns it to `:contact`. If you look back at the *Phone* model, you might be able to guess what this line does: If a `contact` isn't passed to the factory from a spec, it creates a new contact, using the contact factory we've already got, for the test data. Here's a case where it *would* come in handy: Let's verify that two different contacts can share a phone number, perhaps because they share an office or receptionist.

```
# spec/models/phone_spec.rb

it "allows two contacts to share a phone number" do
  FactoryGirl.create(:phone,
    phone_type: "home",
    phone: "785-555-1234")
  FactoryGirl.build(:phone,
    phone_type: "home",
    phone: "785-555-1234").should be_valid
end
```

Of course, validations can be more complicated than just requiring a specific scope. Yours might involve a complex regular expression or a custom method. Get in the habit of testing these validations–not just the happy paths where everything is valid, but also error conditions.

Testing instance methods

It would be convenient to only have to refer to `@contact.name` to render our contacts' full names instead of creating the 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
  FactoryGirl.create(:contact,
    firstname: "John",
    lastname: "Doe").name.should == "John Doe"
end
```

Testing class methods and scopes

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 'spec_helper'

describe Contact do

  # validation examples omitted ...

  it "returns a sorted array of results that match" do
    smith = FactoryGirl.create(:contact, lastname: "Smith")
    jones = FactoryGirl.create(:contact, lastname: "Jones")
    johnson = FactoryGirl.create(:contact, lastname: "Johnson")

    Contact.by_letter("J").should == [johnson, jones]
  end
end
```

Note we're testing the sort order here as well; `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 'spec_helper'

describe Contact do

  # validation examples ...

  it "returns a sorted array of results that match" do
    smith = Factory(:contact, lastname: "Smith")
    jones = Factory(:contact, lastname: "Jones")
    johnson = Factory(:contact, lastname: "Johnson")

    Contact.by_letter("J").should_not 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.

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, `before`, to help simplify my code and reduce typing.

DRYer specs with describe, context, before and after

Like I said, the code above has 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; see below). 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:

```
# spec/models/contact_spec.rb

require 'spec_helper'

describe Contact do

  # validation examples ...

```

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

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

```
# spec/models/contact_spec.rb

require 'spec_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 a function 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

```

```
3  require 'spec_helper'
4
5  describe Contact do
6
7    # validation examples ...
8
9    describe "filter last name by letter" do
10     before :each do
11       @smith = FactoryGirl.create(:contact, lastname: "Smith")
12       @jones = FactoryGirl.create(:contact, lastname: "Jones")
13       @johnson = FactoryGirl.create(:contact, lastname: "Johnson")
14     end
15
16     context "matching letters" do
17       # matching examples ...
18     end
19
20     context "non-matching letters" do
21       # non-matching examples ...
22     end
23   end
24 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 block should be run before *each* example, RSpec will create them for each example individually. In this example, my `before` block 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`.

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` block to clean up after your examples. Since RSpec handles cleaning up the database, I rarely use `after`. `before`, though, is indispensable.

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

```
1  # spec/models/contact_spec.rb
2
3  require 'spec_helper'
4
```

```
5  describe Contact do
6    it "has a valid factory" do
7      FactoryGirl.create(:contact).should be_valid
8    end
9
10   it "is invalid without a firstname" do
11     FactoryGirl.build(:contact, firstname: nil).should_not be_valid
12   end
13
14   it "is invalid without a lastname" do
15     FactoryGirl.build(:contact, lastname: nil).should_not be_valid
16   end
17
18   it "is invalid with a duplicate email address" do
19     FactoryGirl.create(:contact, email: "aaron@everydayrails.com")
20     FactoryGirl.build(:contact,
21       email: "aaron@everydayrails.com").should_not be_valid
22   end
23
24   it "returns a contact's full name as a string" do
25     FactoryGirl.create(:contact,
26       firstname: "John",
27       lastname: "Doe").name.should == "John Doe"
28   end
29
30   describe "filter last name by letter" do
31     before :each do
32       @smith = FactoryGirl.create(:contact, lastname: "Smith")
33       @jones = FactoryGirl.create(:contact, lastname: "Jones")
34       @johnson = FactoryGirl.create(:contact, lastname: "Johnson")
35     end
36
37     context "matching letters" do
38       it "returns a sorted array of results that match" do
39         Contact.by_letter("J").should == [@johnson, @jones]
40       end
41     end
42
43     context "non-matching letters" do
44       it "does not return contacts that start with a different letter" do
45         Contact.by_letter("J").should_not include @smith
46       end
```

```
    end
  end
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
  has a valid factory
  is invalid without a firstname
  is invalid without a lastname
  is invalid with a duplicate email address
  returns a contact's full name as a string
  filter last name by letter
    matching letters
      returns a sorted array of results that match
    non-matching letters
      does not return contacts that don't start with the provided letter

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

Finished in 0.30819 seconds
10 examples, 0 failures
```

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 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. Even better, when we get into testing users with specific roles in chapter 6, might be variables like `@admin_user` and `@guest_user`. *Be expressive with your variable names!*

Summary

And that's 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 *should* and for what *should 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 eight, 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 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.

Questions

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[5] in your application.

Why test the factory itself?

Strictly speaking, this isn't necessary–and as you develop your own testing habits, you may find it unnecessary to verify that your models' factories are yielding what they're supposed to. However, as you continue to use these factories in more complex tests, it can be helpful to know that a failed test is stemming from an issue with your factory and not with something in the functionality at hand.

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

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 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 factory 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.

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. For the most part, his work has been relegated to the education research and instructional technology sectors, which is why you've probably never heard of him until now. When off the clock and away from the text editor, Aaron enjoys photography, baseball (go Cards), college basketball (Rock Chalk Jayhawk), and bowling. He lives with his wife, Elise, along with four cats and a dog in rural Kansas.

Aaron's personal blog is at http://www.aaronsumner.com/; he also blogs about practical usage of Ruby on Rails at Everyday Rails (http://everydayrails.com/). *Everyday Rails Testing with RSpec* is his first book.

Colophon

The cover image of a practical, reliable, red pickup truck[6] is by iStockphoto contributor Habman_18[7]. 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.

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

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