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 2014-02-23

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

Welcome to the Rails 4.0 edition of *Everyday Rails Testing with RSpec*! It's taken a little longer than I'd anticipated, but I hope you'll find the results worth the wait.

I went through from start to finish with an entirely new code base based on the latest and greatest versions of the major players in the book (that is, Rails, RSpec, Capybara, and Factory Girl). I took advantage of Twitter Bootstrap to provide a more plausible example for using Selenium. And, last but not least, I added a new chapter to demonstrate the TDD process by adding a test-driven feature to the sample application. 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.

I want to take a moment to thank, again, everyone who's purchased *Everyday Rails Testing with RSpec* in the past fifteen or so months, or followed the Everyday Rails blog. Believe me when I tell you that the past year has been pretty tumultuous for me, in both bad ways and good. At the risk of sounding too capitalistic, I have to admit that my mood lifted every time I got an email from Leanpub letting me know a new reader was taking a chance on the book.

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

[1] https://github.com/everydayrails/rspec_rails_4/issues

Acknowledgements

First, thank you to why the lucky stiff, wherever he may be, for introducing me to Ruby through his weird, fun projects and books. The Ruby community just isn't the same without him. Thanks to all the other great minds in the Ruby community I haven't met for making me a better developer–even if it doesn't always show in my code.

Thanks to the readers of the Everyday Rails blog for providing good feedback on my original series of RSpec posts, and helping me realize they might make for a decent book. Thanks to everyone who purchased an early copy of the book–the response it's received has been incredible, and your feedback has helped tremendously.

Thanks to David Gnojek for critiquing the dozen or so covers I designed for the book and helping me pick a good one. Check out Dave's work in art and design at DESIGNOJEK[2].

Thanks to family and friends who wished me the best for this project, even though they had no idea what I was talking about.

And finally, thank you to my wife for putting up with my obsession with making new things, even when it keeps me up way too late or awake all night. And thanks to the cats for keeping me company while doing so.

[2] http://www.designojek.com/

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[3] 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 the practice of clicking links in the browser is just too hard to break.

I've been there. I don't consider myself an engineer in the traditional sense, yet just like 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 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[4] 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. Now, to be fair, newer works on Rails have addressed this shortcoming, and now demonstrate how to test applications throughout. 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 adapt to make work consistently for *you*, too.

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

[4] 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 development and basic testing in the likes of Michael Hartl's *Rails Tutorial* (look for the Rails 4-specific version online[5]) 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 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 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.

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

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. 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 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.

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 4.0 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, *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, and time-specific functionality.
- 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

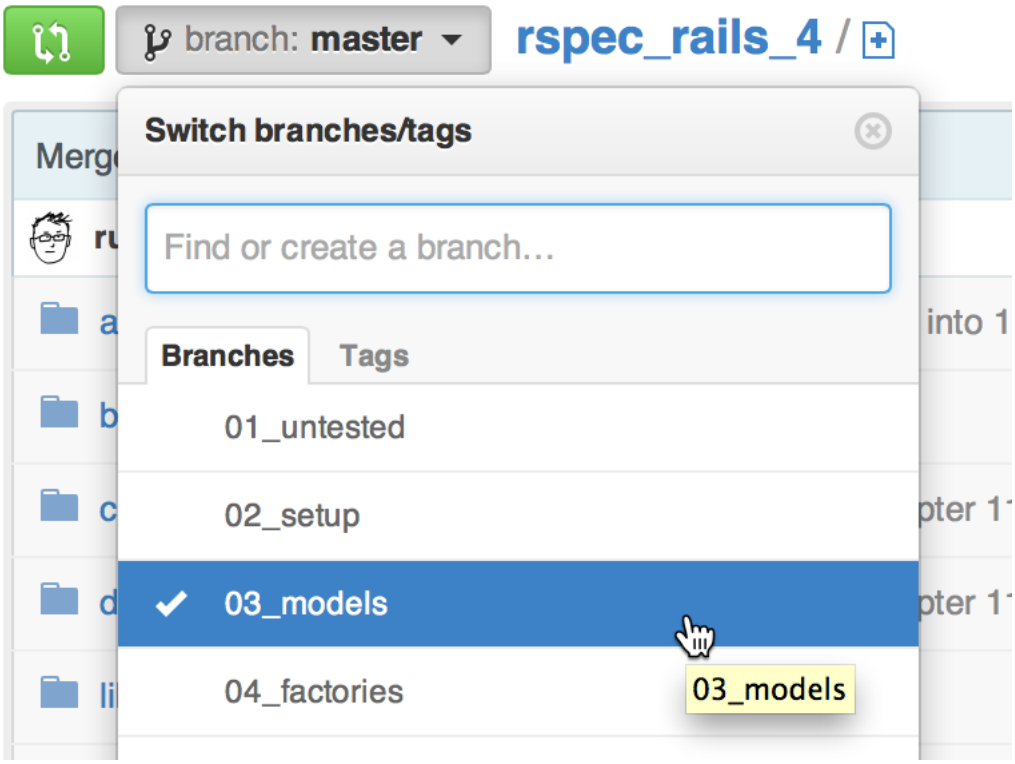
Speaking of the sample code, you can find a completely tested application on GitHub.

Get the source!

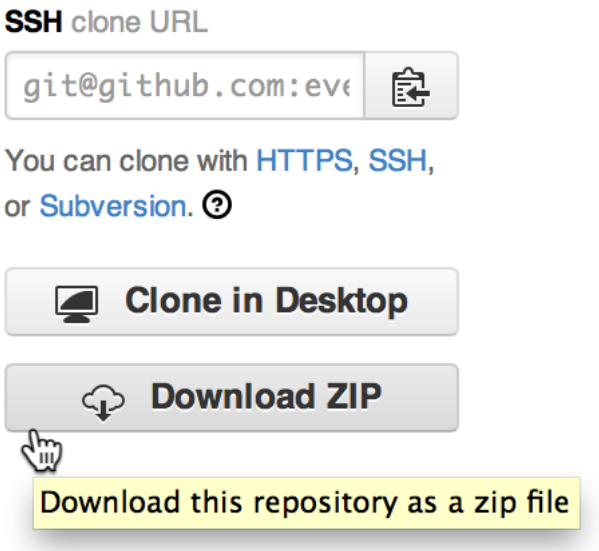
https://github.com/everydayrails/rspec_rails_4

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[6] is an excellent, hands-on way to learn the basics of Git. So is Try Git[7]. For a handy refresher of the basics, check out Git Reference[8].

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

Code conventions

I'm using the following setup for this application:

- **Rails 4.0:** 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 2.0:** I don't think you'll see any major differences if you're using 1.9. At this point I don't recommend trying to progress through the book if you're still using Ruby 1.8.
- **RSpec 2.14:** 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.

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, check the source code for the Rails 3.2 version of the book's project[9]. It's 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[10]–*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/rspec_rails_4/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 changes to existing ones. Finally, users must have an administrator ability to add new users to the system.

[9] https://github.com/ruralocity/everyday_rails_rspec_rails_3_2/
[10] http://learncodethehardway.org/

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, 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.

2. Setting up RSpec

As I mentioned in chapter 1, our contacts manager is currently *functioning*. At least we *think* it's functioning–our only proof of that is we clicked through the links, made a few dummy accounts, and added and edited data. Of course, this doesn't scale as we add features. (I've deployed apps with even less tests than that; I bet some of you have, too.) Before we go any further toward adding new features to the application, we need to stop what we're doing and add an *automated test suite* to it, using RSpec and a few helper gems to make it happen.

Before we dive into those specs, though, we need to do some configuring. Once upon a time, RSpec and Rails took some coaxing to get to work together. That's not really the case anymore, but we'll still need to install a few things and tweak some configurations before we write any specs.

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

- We'll start by using Bundler to install RSpec and other gems useful in testing.
- We'll check for a test database and install one, if necessary.
- Next we'll configure RSpec to test what we want to test.
- Finally, we'll configure a Rails application to automatically generate files for testing as we add new features.

Check out the *02_setup* branch of the sample source to see the completed code for this chapter. Using the command line, type

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

See chapter 1 for additional details.

Gemfile

First things first: Since RSpec isn't included in a default Rails application, we'll need to take a moment to install it and a few helpers. We'll use Bundler to add these dependencies. Let's open our `Gemfile` and add the following code:

Gemfile

```
group :development, :test do
  gem "rspec-rails", "~> 2.14.0"
  gem "factory_girl_rails", "~> 4.2.1"
end

group :test do
  gem "faker", "~> 1.1.2"
  gem "capybara", "~> 2.1.0"
  gem "database_cleaner", "~> 1.0.1"
  gem "launchy", "~> 2.3.0"
  gem "selenium-webdriver", "~> 2.39.0"
end
```

These are the current versions of each gem as I wrote the Rails 4.0 edition of the book, in summer, 2013. Of course, any and all may update frequently, so keep tabs on them on Rubygems.org, GitHub, and your favorite Ruby news feeds.

You need to know Bundler

If this previous code sample is confusing, please set aside this book and find a tutorial on Bundler. (Then come back, please!) Any Rails tutorial covering version 3.0 or newer will have content on Bundler. If you're a Railscasts subscriber, check out the revised version of the Bundler tutorial[11] available there. As it goes with pretty much every other aspect of Rails development these days, we'll be using Bundler heavily in this book.

Why install in two separate groups?

rspec-rails and *factory_girl_rails* are used in both the development and test environments. Specifically, they are used in development by generators we'll be utilizing shortly. The remaining gems are only used when you actually run your specs, so they're not necessary to load in development. This also ensures that gems used solely for generating code or running tests aren't installed in your production environment when you deploy to your server.

Run `bundle install` from your command line to install the gems onto your system.

So what did we just install?

- *rspec-rails* includes RSpec itself in a wrapper to add some extra Rails-specific features.
- *factory_girl_rails* replaces Rails' default fixtures for feeding test data to the test suite with much more preferable *factories*.
- *faker* generates names, email addresses, and other placeholders for factories.

[11] http://railscasts.com/episodes/201-bundler-revised

- *capybara* makes it easy to programatically simulate your users' interactions with your web application.
- *database_cleaner* helps make sure each spec run in RSpec begins with a clean slate, by–you guessed it–cleaning data from the test database.
- *launchy* does one thing, but does it well: It opens your default web browser on demand to show you what your application is rendering. Very useful for debugging tests.
- *selenium-webdriver* will let us test JavaScript-based browser interactions with Capybara.

I'll cover each of these in more detail in future chapters, but in the meantime our application has access to all the basic supports necessary to build a solid test suite. Next up: Creating our test database.

Test database

If you're adding specs to an existing Rails application, there's a chance you've already got a test database on your computer. If not, here's how to add one.

Open the file `config/database.yml` to see which databases your application is ready to talk to. If you haven't made any changes to the file, you should see something like the following if you're using SQLite:

config/database.yml

```
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000
```

Or this if you're using MySQL:

config/database.yml

```
test:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: contacts_test
  pool: 5
  username: root
  password:
  socket: /tmp/mysql.sock
```

Or this if you're using PostgreSQL:

config/database.yml

```
test:
  adapter: postgresql
  encoding: utf8
  database: contacts_test
  pool: 5
  username: root # or your system username
  password:
```

If not, add the necessary code to `config/database.yml` now, replacing *contacts_test* with the appropriate name for your application.

Finally, to ensure there's a database to talk to, run the following rake task:

```
$ bundle exec rake db:create:all
```

If you didn't yet have a test database, you do now. If you did, the rake task politely informs you that the database already exists–no need to worry about accidentally deleting a previous database. Now let's configure RSpec itself.

RSpec configuration

Now we can add a spec folder to our application and add some basic RSpec configuration. We'll install RSpec with the following command line directive:

```
$ bundle exec rails generate rspec:install
```

As the generator dutifully reports, we've now got a configuration file for RSpec (`.rspec`), a directory for our spec files as we create them (`spec`), and a helper file where we'll further customize how RSpec will interact with our code (`spec/spec_helper.rb`).

Next–and this is optional–I like to change RSpec's output from the default format to the easy-to-read *documentation* format. This makes it easier to see which specs are passing and which are failing as your suite runs; it also provides an attractive outline of your specs for–you guessed it–documentation purposes. Open *.rspec* and add the following line:

.rspec

```
--format documentation
```

One last (optional) setup step: Telling Rails to generate spec files for us.

Generators

Thanks to the beauty of Railties[12], just by loading the `rspec-rails` and `factory_girl_rails` gems we're all set. Rails' stock generators will no longer generate the default Test::Unit files in `test`; they'll generate RSpec files in `spec` (and factories in `spec/factories`). However, if you'd like you can manually specify settings for these generators. If you use the `scaffold` generator to add code to your application, you may want to consider this–the default generator adds a lot of specs we won't cover in this book; in particular, view specs.

Open `config/application.rb` and include the following code inside the Application class:

config/application.rb

```
config.generators do |g|
  g.test_framework :rspec,
    fixtures: true,
    view_specs: false,
    helper_specs: false,
    routing_specs: false,
    controller_specs: true,
    request_specs: false
  g.fixture_replacement :factory_girl, dir: "spec/factories"
end
```

Can you guess what this code is doing? Here's a rundown:

- `fixtures: true` specifies to generate a fixture for each model (using a Factory Girl factory, instead of an actual fixture)
- `view_specs: false` says to skip generating view specs. I won't cover them in this book; instead we'll use *feature specs* to test interface elements.
- `helper_specs: false` skips generating specs for the helper files Rails generates with each controller. As your comfort level with RSpec improves, consider changing this option to true and testing these files.
- `routing_specs: false` omits a spec file for your `config/routes.rb` file. If your application is simple, as the one in this book will be, you're probably safe skipping these specs. As your application grows, however, and takes on more complex routing, it's a good idea to incorporate routing specs.
- `request_specs: false` skips RSpec's defaults for adding integration-level specs in *spec/requests*. We'll cover this in chapter 8, at which time we'll just create our own files.
- And finally, `g.fixture_replacement :factory_girl` tells Rails to generate factories instead of fixtures, and to save them in the `spec/factories` directory.

Don't forget, just because RSpec won't be generating some files for you doesn't mean you can't add them by hand, or delete any generated files you're not using. For example, if you need to add

[12] http://api.rubyonrails.org/classes/Rails/Railtie.html

a helper spec, just add it inside *spec/helpers*, following the spec file naming convention. So if we wanted to test *app/helpers/contacts_helper.rb*, we'd add *spec/helpers/contacts_helper_spec.rb*. If we wanted to test a hypothetical library in *lib/my_library.rb* we'd add a spec file *spec/lib/my_library_spec.rb*. And so on.

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:

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

This *clones* the database structure as used in development to the test database. 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.

ABC: Always Be Cloning

Don't forget, any time you use `rake db:migrate` to 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.*

You can chain the two rake tasks together on your command line with `rake db:migrate db:test:clone`, or you can go one step further and create a shell alias with a shortcut. For example, I use the shortcut `rmigc` to run a migration and clone the database with a single command.

With that, our application is now configured to test with RSpec! We can even give it a first run:

```
$ bundle exec rspec

No examples found.

Finished in 0.00013 seconds
0 examples, 0 failures
```

Looks good! In the next chapter we'll start using it to actually test the application's functionality, starting with its model layer.

Questions

- **Can I delete my *test* folder?** If you're starting a new application from scratch, yes. If you've been developing your application for awhile, first run `rake test` to verify that there aren't any tests contained within the directory that you may want to transfer to RSpec's spec folder.
- **Why don't you test views?** Trust me, creating reliable view tests is a hassle. Maintaining them is even worse. As I mentioned when I set up my generators to crank out spec files, I relegate testing UI-related code to my integration tests. This is a pretty standard practice among Rails developers.

Exercises

If you're working from an existing code base:

- Add RSpec and the other required gems to your `Gemfile`, and use Bundler to install. The code and techniques provided in this book will work with Rails 3.0 and newer.
- Make sure your application is properly configured to talk to your test database. Create your test database, if necessary.
- Go ahead and configure Rails' generator command to use RSpec and Factory Girl for any new application code you may add moving forward. You can also just use the default settings provided by the gems.
- Make a list of things you need to test in your application as it now exists. This can include mission-critical functionality, bugs you've had to track down and fix in the past, new features that broke existing features, or edge cases to test the bounds of your application. We'll cover these scenarios in the coming chapters.

If you're working from a new, pristine code base:

- Follow the instructions for installing RSpec and associates with Bundler.
- Your *database.yml* file should already be configured to use a test database. If you're using a database besides SQLite you'll probably need to create the actual database, if you haven't already, with `bundle exec rake db:create:all`.
- Optionally, configure Rails' generators to use RSpec and Factory Girl, so that as you add new models and controllers to your application you'll automatically be given starter files for your specs and factories.

Extra credit:

OK, I'm not actually handing out points for any of this–but if you create a lot of new Rails applications, you can create a Rails application template to automatically add RSpec and related configuration to your `Gemfile` and application config files, not to mention create your test database. Rails Wizard[13] and App Scrolls[14] are great starting points for building application templates of your favorite tools.

[13] http://railswizard.org/
[14] http://appscrolls.org

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
```

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.
- **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 'spec_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 'spec_helper'` at the top, and get used to typing it–all of your specs will include this line moving forward.

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 (using `bundle exec rspec`) the output would be similar to the following:

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

Finished in 0.00098 seconds
```

```
6 examples, 0 failures, 6 pending

Randomized with seed 32236
```

Great! Six pending specs–let's write them and make them pass, starting with the first example. Before that, though, we need to look at a recent, important change to the RSpec DSL.

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.)

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[15]. Instead of saying something `should` or `should_not` match expected output, you `expect` something `to` or `to_not` be something else.

As an example, let's look at this sample expectation. In Ruby, `true` should always, well, *be true*. In the old RSpec syntax, this would be written like this:

```
it "is true when true" do
  true.should be_true
end
```

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

```
it "is true when true" do
  expect(true).to be_true
end
```

Even though many RSpec tutorials still use the old `should` syntax, we'll only use the newer `expect()` format here. It'll help you build the muscle memory now instead of later–and it's possible that `should` will require additional configuration in future versions of RSpec.

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:

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

spec/models/contact_spec.rb

```
require 'spec_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

  # remaining examples to come
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 `bundle exec rspec`) we see one passing example! We're on our way. Now let's get into testing more of our code.

If RSpec reports that it `could not find table 'contacts'`, remember to run the `rake db:test:clone` task to add your application's tables to the test database.

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

```
it "is invalid without a firstname" do
  expect(Contact.new(firstname: nil)).to have(1).errors_on(:firstname)
end
```

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

spec/models/contact_spec.rb

```
it "is invalid without a firstname" do
  expect(Contact.new(firstname: nil)).to_not have(1).errors_on(:firstname)
end
```

And sure enough, RSpec reports a failure:

```
Failures:

  1) Contact is invalid without a firstname
     Failure/Error: expect(Contact.new(firstname: nil)).to_not
      have(1).errors_on(:firstname)
       expected target not to have 1 errors_on, got 1
     # ./spec/models/contact_spec.rb:13:in `block (2 levels) in
     <top (required)>'
```

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 `to_not` back to `to` before continuing.

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

spec/models/contact_spec.rb

```
it "is invalid without a lastname" do
  expect(Contact.new(lastname: nil)).to have(1).errors_on(:lastname)
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')
  expect(contact).to have(1).errors_on(:email)
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

```
1  require 'spec_helper'
2
3  describe Phone do
4    it "does not allow duplicate phone numbers per contact" do
5      contact = Contact.create(firstname: 'Joe', lastname: 'Tester',
6        email: 'joetester@example.com')
7      contact.phones.create(phone_type: 'home',
8        phone: '785-555-1234')
9      mobile_phone = contact.phones.build(phone_type: 'mobile',
10       phone: '785-555-1234')
11
12     expect(mobile_phone).to have(1).errors_on(:phone)
13   end
14
15   it "allows two contacts to share a phone number" do
16     contact = Contact.create(firstname: 'Joe', lastname: 'Tester',
17       email: 'joetester@example.com')
18     contact.phones.create(phone_type: 'home',
19       phone: '785-555-1234')
20     other_contact = Contact.new
21     other_phone = other_contact.phones.build(phone_type:
22       'home', phone: '785-555-1234')
23
24     expect(other_phone).to be_valid
25   end
26 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

```
1 def name
2   [firstname, lastname].join(' ')
3 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

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

RSpec prefers `eq` over `==` 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 'spec_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 'spec_helper'

describe Contact do

  # validation examples ...

  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_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.

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[16]. 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:

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

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 even 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 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

```
require 'spec_helper'

describe Contact do

  # validation examples ...

  describe "filter last name by letter" do
    before :each 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')
    end

    context "matching letters" do
      # matching examples ...
    end

    context "non-matching letters" do
      # non-matching examples ...
    end
  end
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 'spec_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
    expect(Contact.new(firstname: nil)).to have(1).errors_on(:firstname)
  end

  it "is invalid without a lastname" do
    expect(Contact.new(lastname: nil)).to have(1).errors_on(:lastname)
  end

  it "is invalid without an email address" do
    expect(Contact.new(email: nil)).to have(1).errors_on(:email)
  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',
        email: 'tester@example.com')
    expect(contact).to have(1).errors_on(:email)
  end

  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

  describe "filter last name by letter" do
    before :each do
      @smith = Contact.create(firstname: 'John', lastname: 'Smith',
        email: 'jsmith@example.com')
```

```
44        @jones = Contact.create(firstname: 'Tim', lastname: 'Jones',
45          email: 'tjones@example.com')
46        @johnson = Contact.create(firstname: 'John', lastname: 'Johnson',
47          email: 'jjohnson@example.com')
48      end
49
50      context "matching letters" do
51        it "returns a sorted array of results that match" do
52          expect(Contact.by_letter("J")).to eq [@johnson, @jones]
53        end
54      end
55
56      context "non-matching letters" do
57        it "returns a sorted array of results that match" do
58          expect(Contact.by_letter("J")).to_not include @smith
59        end
60      end
61    end
62  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
  returns a contact's full name as a string
  is invalid without a firstname
  is invalid with a duplicate email address
  is invalid without a lastname
  is valid with a firstname, lastname and email
  is invalid without an email address
  filter last name by letter
    matching letters
      returns a sorted array of results that match
    non-matching letters
      returns a sorted array of results that match

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

Finished in 0.44718 seconds
10 examples, 0 failures

Randomized with seed 28299
```

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 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[17] 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.

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

4. Generating test data with factories

So far we've been using *plain old Ruby objects* to create temporary data for our tests. And so far, our tests haven't been so complex that much more than that has been necessary. As we test more complex scenarios, though, it sure would be nice to simplify that aspect of the process and focus more on the *test* instead of the *data*. Luckily, a handful of Ruby libraries exist to make test data generation easy. In this chapter we'll focus on Factory Girl, the preferred approach for many developers. Specifically:

- We'll talk about the benefits and drawbacks of using factories as opposed to other methods.
- Then we'll create a basic factory and apply it to our existing specs.
- Following that we'll edit our factories to make them even more convenient to use.
- Next we'll create more realistic test data using the Faker gem.
- We'll look at more advanced factories relying on Active Record associations.
- Finally, we'll talk about the risks of taking factory implementation too far in your applications.

Check out the *04_factories* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 04_factories origin/04_factories
```

See chapter 1 for additional details.

If you haven't done so already, make sure you've got the *factory_girl_rails* and *faker* gems installed in your application, as outlined in chapter 2.

Factories versus fixtures

Out of the box, Rails provides a means of quickly generating sample data called *fixtures*. A fixture is essentially a YAML-formatted file which helps create sample data. For example, a fixture for our `Contact` model might look like

contacts.yml

```
aaron:
  firstname: "Aaron"
  lastname: "Sumner"
  email: "aaron@everydayrails.com"

john:
  firstname: "John"
  lastname: "Doe"
  email: "johndoe@nobody.org"
```

Then, by referencing `contacts(:aaron)` in a test, I've instantly got a fresh `Contact` with all attributes set. Pretty nice, right?

Fixtures have their place, but also have their drawbacks. 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[18], an easy-to-use and easy-to-rely-on gem for creating test data without the brittleness of fixtures.

Of course, the Ruby community is always up for a good debate on best practices, and Factory Girl also has its naysayers. In summer of 2012 an online debate over the merit of factories[19] sprung up. A number of vocal opponents, including Rails' creator David Heinemeier Hansson, pointed out that factories are a primary cause of slow test suites, and that factories can be particularly cumbersome with complex associations.

While I see their point and acknowledge that the ease of using factories can come with a cost in terms of speed, I still believe that a slow test is better than no test, and that a factory-based approach simplifies things for people who are just learning how to test to begin with. You can always swap out factories for more efficient approaches later once you've got a suite built and are more comfortable with testing.

In the meantime, let's put factories to work in our application. Since the *factory_girl_rails* gem installed Factory Girl for us as a dependency (see chapter 2), we're ready to roll.

Adding factories to the application

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

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

[19] https://groups.google.com/forum/?fromgroups#!topic/rubyonrails-core/_lcjRRgyhC0

spec/factories/contacts.rb

```
FactoryGirl.define do
  factory :contact do
    firstname "John"
    lastname "Doe"
    sequence(:email) { |n| "johndoe#{n}@example.com"}
  end
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. (Later in this chapter, we'll look at a nice alternative to generating things like email addresses and names, called Faker.)

Although this example's attributes are all strings, you're not limited to strings-only in your own factories. You can pass along whatever your attribute's data type expects to see, including integers, booleans, and dates. You can even pass in Ruby code to dynamically assign values; just remember to do so within a block (as shown in the sequence example above). For example, if we stored our contacts' birthdays, we could easily generate those dates from factories by using Ruby datetime methods such as `33.years.ago` or `Date.parse('1980-05-13')`.

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 of the model it corresponds to (so, *spec/factories/contacts.rb* for the Contact model). I tend to just stick with that approach, too. Bottom line: As long as your factory definitions are syntactically correct and located in *spec/factories/*, you should be fine.

With a solid factory in place, let's return to the *contact_spec.rb* file we set up in the previous chapter and add a quick example to it:

spec/models/contact_spec.rb

```
require 'spec_helper'

describe Contact do
  it "has a valid factory" do
    expect(FactoryGirl.build(:contact)).to be_valid
  end

  ## more specs
end
```

This instantiates (but does not save) a new contact with attributes as assigned by the factory. It then tests that new contact's validity. Compare that to our spec from the previous chapter, which required including all required attributes to pass:

```
1  it "is valid with a firstname, lastname and email" do
2    contact = Contact.new(
3      firstname: 'Aaron',
4      lastname: 'Sumner',
5      email: 'tester@example.com')
6    expect(contact).to be_valid
7  end
```

Let's revisit our existing specs, now using Factory Girl to streamline building our data. This time we'll override one or more attributes to generate data from factories, but with specific attributes:

spec/models/contact_spec.rb

```
it "is invalid without a firstname" do
  contact = FactoryGirl.build(:contact, firstname: nil)
  expect(contact).to have(1).errors_on(:firstname)
end

it "is invalid without a lastname" do
  contact = FactoryGirl.build(:contact, lastname: nil)
  expect(contact).to have(1).errors_on(:lastname)
end

it "is invalid without an email address" do
  contact = FactoryGirl.build(:contact, email: nil)
  expect(contact).to have(1).errors_on(:email)
end

it "returns a contact's full name as a string" do
  contact = FactoryGirl.build(:contact,
    firstname: "Jane", lastname: "Doe")
  expect(contact.name).to eq "Jane Doe"
end
```

These examples are pretty straightforward. As in our earlier example, all use Factory Girl's `build` method to create a new, yet non-persisted, `Contact`. The first example's spec assigns `contact` to a `Contact` with no `firstname` assigned. The second follows suit, replacing the factory's default `lastname` with nil. Since our `Contact` model validates presence of both `firstname` and `lastname`, both of these examples expect to see errors. Follow the same pattern to test the validation for `email`.

The fourth spec is a little different, but uses the same basic tools. This time, we're creating a new `Contact` with specific values for `firstname` and `lastname`. Then, we're making sure that the `name` method on the assigned `contact` returns the string we expect.

The next spec throws in a minor wrinkle:

spec/models/contact_spec.rb

```
it "is invalid with a duplicate email address" do
  FactoryGirl.create(:contact, email: "aaron@example.com")
  contact = FactoryGirl.build(:contact, email: "aaron@example.com")
  expect(contact).to have(1).errors_on(:email)
end
```

In this example, we're making sure the test object's `email` attribute is not duplicate data. In order to do this, we need another `Contact` persisted in the database–so before running the expectation, we use `FactoryGirl.create` to first persist a `contact` with the same email address.

Simplifying our syntax

Most programmers I know hate typing any more than they have to. And typing `FactoryGirl.build(:contact)` each time we need a new contact is already getting cumbersome. Luckily, Factory Girl version 3.0 and newer makes the Rails programmer's life a bit simpler with a little configuration. Add it anywhere inside the the `RSpec.configure` block located in *spec_helper.rb*:

spec/spec_helper.rb

```
1  RSpec.configure do |config|
2    # Include Factory Girl syntax to simplify calls to factories
3    config.include FactoryGirl::Syntax::Methods
4
5    # other configurations omitted ...
6  end
```

Now our specs can use the shorter `build(:contact)` syntax. This one line of configuration also gives us `create(:contact)`, which we've already used; and `attributes_for(:contact)` and `build_stubbed(:contact)`, which we'll use in subsequent chapters.

Here's a look at our updated, leaner model spec:

spec/models/contact_spec.rb

```
require 'spec_helper'

describe Contact do
  it "has a valid factory" do
    expect(build(:contact)).to be_valid
  end

  it "is invalid without a firstname" do
    expect(build(:contact, firstname: nil)).to \
      have(1).errors_on(:firstname)
  end

  it "is invalid without a lastname" do
    expect(build(:contact, lastname: nil)).to \
      have(1).errors_on(:lastname)
  end

  # remaining examples omitted ...
end
```

Much more readable, if you ask me, but entirely optional in your own code.

Associations and inheritance in factories

If we were to create a factory for our `Phone` model, given what we know so far, it might look something like this.

spec/factories/phones.rb

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

New here is the call to `:association`; that tells Factory Girl to create a new `Contact` on the fly for this phone to belong to if one wasn't passed into the `build` (or `create`) method.

However, a contact can have three types of phones–home, office, and mobile. So far, if we wanted to specify a non-home phone in a spec we've done it like this:

spec/models/phone_spec.rb

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

Let's do some refactoring to clean this up. Factory Girl provides us the ability to create *inherited* factories, overriding attributes as necessary. In other words, if we specifically want an office phone in a spec, we should be able to call it with `build(:office_phone)` (or the longer `FactoryGirl.build(:office_phone)`, if you prefer). Here's how it looks:

spec/factories/phones.rb

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

    factory :home_phone do
      phone_type 'home'
    end

    factory :work_phone do
      phone_type 'work'
    end

    factory :mobile_phone do
      phone_type 'mobile'
    end
  end
end
```

And the spec can be simplified to

spec/models/phone_spec.rb

```
require 'spec_helper'

describe Phone do
  it "does not allow duplicate phone numbers per contact" do
    contact = create(:contact)
    create(:home_phone,
      contact: contact,
      phone: '785-555-1234')
    mobile_phone = build(:mobile_phone,
      contact: contact,
      phone: '785-555-1234')
    expect(mobile_phone).to have(1).errors_on(:phone)
  end

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

This technique will come in handy in subsequent chapters when we need to create different user types (administrators versus non-administrators) for testing authentication and authorization mechanisms.

Generating more realistic fake data

Earlier in this chapter, we used a *sequence* to make sure the contacts factory yielded unique email addresses. We can improve on this by providing more realistic test data to our app, using a fake data generator called–what else?–*Faker*. Faker is a Ruby port of a time-honored Perl library for generating fake names, addresses, sentences, and more–excellent for testing purposes.

Let's incorporate some fake data into our factories:

spec/factories/contacts.rb

```
require 'faker'

FactoryGirl.define do
  factory :contact do
    firstname { Faker::Name.first_name }
    lastname { Faker::Name.last_name }
    email { Faker::Internet.email }
  end
end
```

Now our specs will use a random email address each time the phone factory is used. (To see for yourself, check out *log/test.log* after running specs to see the email addresses that were inserted into the database in *contact_spec.rb*.) Two important things to observe here: First, we've required the Faker library to load in the first line of my factory; and second, that we pass the `Faker::Internet.email` method inside a block–Factory Girl considers this a "lazy attribute" as opposed to the statically-added string the factory previously had.

Let's wrap up this exercise by returning to that phone factory. Instead of giving every new phone a default number, let's give them all unique, random, realistic ones:

spec/factories/phones.rb

```
require 'faker'

FactoryGirl.define do
  factory :phone do
    association :contact
    phone { Faker::PhoneNumber.phone_number }

    # child factories omitted ...
  end
end
```

Yes, this isn't strictly necessary. I could keep using sequences and my specs would still pass. But Faker does give us a bit more realistic data with which to test (not to mention, some of the data generated by Faker can be pretty fun).

Faker can generate other types of random data such as addresses, phony business names and slogans, and *lorem* placeholder text–refer to the documentation[20] for more.

Check out Forgery[21] as an alternative to Faker. Forgery performs a similar function but has a bit different syntax. There's also ffaker[22], a rewrite of Faker running up to 20 times faster than the original. And these gems aren't just useful for testing–see how to use Faker to obfuscate data for screenshots[23] in the Everyday Rails blog.

Advanced associations

The validation specs we've created so far have, for the most part, tested relatively simple aspects of our data. They haven't required us to look at anything but the models in question–in other words, we haven't validated that when we create a contact, three phone numbers also get created. How do we test that? And how do we make a factory to make sure our test contacts continue to represent realistic ones?

[20] http://rubydoc.info/gems/faker/1.0.1/frames
[21] https://github.com/sevenwire/forgery
[22] https://github.com/emmanueloga/ffaker
[23] http://everydayrails.com/2013/05/20/obfuscated-data-screenshots.html

The answer is to use Factory Girl's *callbacks* to add additional code to a given factory. Callbacks are particularly useful in testing nested attributes, as in the way our user interface allows phone numbers to be entered upon creating or editing a contact. For example, this modification to our contact factory uses the `after` callback to make sure that a new contact built with the factory will also have one each of the three phone types assigned to it:

spec/factories/contacts.rb

```
require 'faker'

FactoryGirl.define do
  factory :contact do
    firstname { Faker::Name.first_name }
    lastname { Faker::Name.last_name }
    email { Faker::Internet.email }

    after(:build) do |contact|
      [:home_phone, :work_phone, :mobile_phone].each do |phone|
        contact.phones << FactoryGirl.build(:phone,
          phone_type: phone, contact: contact)
      end
    end
  end
end
```

Note that `after(:build)` takes a block, and within that block, an array of our three phone types is used to also build a contact's phone numbers. We can make sure this is working with the following example:

spec/models/contact_spec.rb

```
it "has three phone numbers" do
  expect(create(:contact).phones.count).to eq 3
end
```

This example passes, and existing examples pass as well, so changing the factory didn't break any of our existing work. We can even take this a step further, and add a validation inside the Contact model itself to make sure this happens:

app/models/contact.rb

```
validates :phones, length: { is: 3 }
```

As an experiment, try changing the value in the validation to some other number, and run the test suite again. All of the examples that were expecting a valid contact will fail. As a second

experiment, comment out the `after` block in the contact factory and run the test suite–again, a whole lot of red.

While our example is specific to the nature of a contact's three phone numbers, Factory Girl callbacks are by no means as limited. Check out the post Get Your Callbacks On with Factory Girl 3.3[24] from Thoughtbot, for more on how to take advantage of this feature.

While this example may seem somewhat contrived, it does represent something you'll sooner or later encounter in a complex application. In fact, this example is based on a scheduling system I once built, requiring a user to add a minimum of two attendees to a meeting. It took me awhile to dig through the Factory Girl documentation, code, and Internet at large to get my factories working correctly with this requirement.

`after(:build)` is just one callback now at our disposal–as you might guess, we can also use `before(:build)`, `before(:create)`, and `after(:create)`. They all work similarly.

How to abuse factories

Factories are great, except when they're not. As mentioned at the beginning of this chapter, unchecked factory usage can cause a test suite to slow down in a hurry–especially when the complexities of associations are introduced. In fact, I'd say that our last factory's creation of three additional objects every time it is called is pushing it–but at least at this point the convenience of generating that data with one method call instead of several outweighs any drawbacks.

While generating associations with factories is an easy way to ramp up tests, it's also an easy feature to abuse and often a culprit when test suites' running times slow to a crawl. When that happens, it's better to remove associations from factories and build up test data manually. You can also fall back to the *Plain Old Ruby Objects* approach we used in chapter 3, or even a hybrid approach combining them with factories.

If you've looked at other resources for testing in general or RSpec specifically, you've no doubt run across the terms *mocks* and *stubs*. If you've already got a bit of testing experience under your belt, you may wonder why I've been using factories all this time and not mocks and stubs. The answer is because, from my experience, basic objects and factories are easier for getting developers started and comfortable with testing–not to mention, overuse of mocks and stubs can lead to a separate set of problems.

Since at this stage our application is pretty small, any speed increase we'd see with a fancier approach would be negligible. That said, mocks and stubs do have their roles in testing; we'll talk more about them in chapters 9 and 10.

Summary

Factory Girl's been of good use to us in this chapter. We've now got less syntax to clutter up our specs, a flexible way to create specific types of data, more realistic fake data, and a way to build

[24] http://robots.thoughtbot.com/post/23039827914/get-your-callbacks-on-with-factory-girl-3-3

more complex associations as needed. What you now know should get you through most testing tasks, but refer also to Factory Girl's documentation[25] for additional usage examples–Factory Girl could almost warrant its own short book.

And while it's not perfect, we'll be using Factory Girl throughout the remainder of the book–the convenience it provides as we become more proficient in testing outweighs the issue of speed. In fact, it will play an important role in testing our next batch of code: The controllers that keep data moving between models and views. That will be the focus of the next chapter.

Exercises

- Add factories to your application, if you haven't done so already.
- Configure RSpec to use the shorter Factory Girl syntax in specs. How does doing so affect the readability of your examples?
- Take a look at your application's factories. How can you refactor them with inherited factories?
- Do your models lend themselves to data types supported by Faker? Take another look at the Faker documentation if necessary, then apply Faker methods to your factories where applicable. Do your specs still pass?
- Do any models in your application use nested attributes? Would using the `after(:build)` callback result in more realistic test data?

[25] https://github.com/thoughtbot/factory_girl/blob/master/GETTING_STARTED.md

5. Basic controller specs

Poor controllers. As Rails developers we keep them skinny (which is a good thing) and often don't give them due attention in our tests (which can be a bad thing; more on that in a moment). As you continue to improve your application's test coverage, though, controllers are the next logical chunk of code to tackle.

Part of the challenge of testing controllers is they can be dependent on a number of other factors–how your models are configured to relate to one another, for example, or how you have your application's routing set up. Hang on, we're going to address some of these challenges in this chapter–but once you've made it through you'll have a clearer understanding of how to build controller specs in your own software.

In this chapter, we'll begin covering a little more ground:

- First, we'll discuss why you should test controllers at all.
- We'll follow that discussion with the very basics (or, *controller specs are just unit specs*).
- Next we'll begin organizing controller specs in an outline-like format.
- We'll then use factories to set up data for specs.
- Then we'll test the seven CRUD methods included in most controllers, along with a non-CRUD example.
- Next, we'll look at testing nested routes.
- We'll wrap up with testing a controller method with non-HTML output, such as a file export.

Check out the *05_controller_basics* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 05_controller_basics origin/05_controller_basics
```

See chapter 1 for additional details.

For this exercise, we need to make a slight modification to our application's `contacts_controller.rb` file. In order to focus on the basics of controller testing, we need to bypass the application's authentication layer for the duration of the chapter. The quickest way to do that is to comment it out:

app/controllers/contacts_controller.rb

```
1 class ContactsController < ApplicationController
2   # before_action :authenticate, except: [:index, :show]
3   before_action :set_contact, only: [:show, :edit, :update, :destroy]
4
5   # etc.
```

Why test controllers?

There are a few good reasons to explicitly test your controller methods:

- **Controllers are classes with methods, too**, as Piotr Solnica indicated in an excellent blog post[26]. And in Rails applications, they're pretty important classes (and methods)–so it's a good idea to put them on equal footing, spec-wise, as your Rails models.
- **Controller specs can often be written more quickly than their integration spec counterparts**. For me, this becomes critical when I encounter a bug that's residing at the controller level, or I want to add additional specs to verify some refactoring. Writing a solid controller spec is a comparatively straightforward process, since I can generate very specific input to the method I'm testing without the overhead of feature specs. This also means that
- **Controller specs usually run more quickly than integration specs**, making them very valuable during bug fixing and checking the bad paths your users can take (in addition to the good ones, of course).

Why not test controllers?

So why is it you don't see controller specs used heavily in many open source Rails projects? Here are some thoughts:

- **Controllers should be skinny**–so skinny, some suggest, that testing them is fruitless.
- **Controller specs, while faster than feature specs, are still slower than specs of Rails models and plain Ruby objects**. This will be mitigated somewhat when we look at ways to speed up our specs in chapter 9, but it's a very valid point.
- **One feature spec can accomplish the work of multiple controller specs**–so maybe it's simpler to write and maintain a single spec instead of several.

In the end, I suspect, the true answer lies somewhere in the middle. In earlier editions of this book, I talked about my own ongoing internal struggle with controller specs. When I was learning RSpec and TDD, understanding controller specs were integral to my overall understanding of the tools and the process. And that's why I want to take a long look at them here: They're a great way to practice using a lot of RSpec features we typically wouldn't use at the model or feature layers. In addition, they *are* still good for testing controller nuances without the overhead of feature specs.

[26] http://solnic.eu/2012/02/02/yes-you-should-write-controller-tests.html

Controller testing basics

Scaffolds, when done correctly, are a great way to learn coding techniques. The spec files generated for controllers, at least as of RSpec 2.8, are pretty nice and provide a good template to help you build your own specs. Look at the scaffold generator in rspec-rails' source[27], or generate a scaffold in your properly-configured-for-RSpec-Rails application to begin getting a sense of these tests. (Another generator to look at is the one in Nifty Generator's scaffolds[28]).

A controller spec is broken down by controller method–each example is based off of a single action and, optionally, any parameters passed to it. Here's a simple example:

```
it "redirects to the home page upon save" do
  post :create, contact: FactoryGirl.attributes_for(:contact)
  expect(response).to redirect_to root_url
end
```

You may spot similarities to earlier specs we've written:

- The description of the example is written in *explicit, active language.*
- *The example only expects one thing*: After the post request is processed, a redirect should be returned to the browser.
- *A factory generates test data to pass to the controller method*; note the use of Factory Girl's `attributes_for` option, which generates a hash of values as opposed to a Ruby object. Yes, you can provide a plain old hash without invoking an extra dependency; however, for convenience we'll stick with Factory Girl.

However, there are also a couple of new things to look at:

- *The basic syntax of a controller spec*–its HTTP method (`post`), controller method (`:create`), and, optionally, parameters being passed to the method.
- *The aforementioned `attributes_for` call to Factory Girl*–not rocket science, but worth mentioning again because I had a habit early on of forgetting to use it versus default factories. As a reminder, `attributes_for()` generates a hash of attributes, not an object.

Organization

Let's start with a top-down approach. As I mentioned earlier during our look at model specs, it's helpful to think about a spec as an outline of things we need our Ruby class to do. We'll start with a spec for our sample application's contacts controller (again, ignoring authorization for now):

[27] https://github.com/rspec/rspec-rails/tree/master/lib/generators/rspec/scaffold
[28] https://github.com/ryanb/nifty-generators/tree/master/rails_generators/nifty_scaffold

spec/controllers/contacts_controller_spec.rb

```
require 'spec_helper'

describe ContactsController do

  describe 'GET #index' do
    context 'with params[:letter]' do
      it "populates an array of contacts starting with the letter"
      it "renders the :index template"
    end

    context 'without params[:letter]' do
      it "populates an array of all contacts"
      it "renders the :index template"
    end
  end

  describe 'GET #show' do
    it "assigns the requested contact to @contact"
    it "renders the :show template"
  end

  describe 'GET #new' do
    it "assigns a new Contact to @contact"
    it "renders the :new template"
  end

  describe 'GET #edit' do
    it "assigns the requested contact to @contact"
    it "renders the :edit template"
  end

  describe "POST #create" do
    context "with valid attributes" do
      it "saves the new contact in the database"
      it "redirects to contacts#show"
    end

    context "with invalid attributes" do
      it "does not save the new contact in the database"
      it "re-renders the :new template"
    end
  end

  describe 'PATCH #update' do
```

```
45      context "with valid attributes" do
46        it "updates the contact in the database"
47        it "redirects to the contact"
48      end
49
50      context "with invalid attributes" do
51        it "does not update the contact"
52        it "re-renders the #edit template"
53      end
54    end
55
56    describe 'DELETE #destroy' do
57      it "deletes the contact from the database"
58      it "redirects to users#index"
59    end
60  end
```

As in our model specs, we can use RSpec's `describe` and `context` blocks to organize examples into a clean hierarchy, based on a controller's actions and the context we're testing–in this case, the happy path (a method received valid attributes to the controller) and the unhappy path (a method received invalid or incomplete attributes).

Setting up test data

Just as in model specs, controller specs need data. Here again we'll use factories to get started–once you've got the hang of it you can swap these out with more efficient means of creating test data, but for our purposes (and this small app) factories will work great.

Here's the factory we already created for contacts; let's add to it to include an *invalid* contact child factory:

spec/factories/contacts.rb

```
1  require 'faker'
2
3  FactoryGirl.define do
4    factory :contact do
5      firstname { Faker::Name.first_name }
6      lastname { Faker::Name.last_name }
7      email { Faker::Internet.email }
8
9      after(:build) do |contact|
10       [:home_phone, :work_phone, :mobile_phone].each do |phone|
11         contact.phones << FactoryGirl.build(:phone,
12           phone_type: phone, contact: contact)
13       end
```

```
14      end
15
16      factory :invalid_contact do
17        firstname nil
18      end
19    end
20  end
```

Remember how we used factory inheritance to create a `:home_phone`, `:office_phone`, and `:mobile_phone` from a parent `:phone` factory? We can use that same technique to create an `:invalid_contact` from the base `:contact` factory. It replaces the specified attributes (in this case, `firstname`) with its own; everything else will defer to the original `:contact` factory.

Testing GET requests

A standard, CRUD-based Rails controller is going to have four GET-based methods: *index*, *show*, *new*, and *edit*. These methods are generally the easiest to test, as they should only be returning data from the browser. In the interest of simplicity, let's start with *show*:

spec/controllers/contacts_controller_spec.rb

```
1  describe 'GET #show' do
2    it "assigns the requested contact to @contact" do
3      contact = create(:contact)
4      get :show, id: contact
5      expect(assigns(:contact)).to eq contact
6    end
7
8    it "renders the :show template" do
9      contact = create(:contact)
10     get :show, id: contact
11     expect(response).to render_template :show
12   end
13 end
```

Let's break this down. We're checking for two things here: First, that a persisted contact is found by the controller method and properly assigned to the specified instance variable. To accomplish this, we're taking advantage of the `assigns()` method–checking that the value (*assigned* to `@contact`) is what we expect to see.

The second expectation may be self-explanatory, thanks to RSpec's clean, readable syntax: The response sent from the controller back up the chain toward the browser will be rendered using the `show.html.erb` template.

These two simple expectations demonstrate the following key concepts of controller testing:

- The basic DSL for interacting with controller methods: Each HTTP verb has its own method (in these cases, `get`), which expects the controller method name as a symbol (here, `:show`), followed by any params (`id: contact`).
- Variables instantiated by the controller method can be evaluated using `assigns(:variable_name)`.
- The finished product returned from the controller method can be evaluated through `response`.

Now let's visit the slightly trickier *index* method.

spec/controllers/contacts_controller_spec.rb

```
1  describe 'GET #index' do
2    context 'with params[:letter]' do
3      it "populates an array of contacts starting with the letter" do
4        smith = create(:contact, lastname: 'Smith')
5        jones = create(:contact, lastname: 'Jones')
6        get :index, letter: 'S'
7        expect(assigns(:contacts)).to match_array([smith])
8      end
9
10     it "renders the :index template" do
11       get :index, letter: 'S'
12       expect(response).to render_template :index
13     end
14   end
15
16   context 'without params[:letter]' do
17     it "populates an array of all contacts" do
18       smith = create(:contact, lastname: 'Smith')
19       jones = create(:contact, lastname: 'Jones')
20       get :index
21       expect(assigns(:contacts)).to match_array([smith, jones])
22     end
23
24     it "renders the :index template" do
25       get :index
26       expect(response).to render_template :index
27     end
28   end
29 end
```

Let's break this down, starting with the first context. We're checking for two things here: First, that an array of contacts matching the first-letter search is created and assigned to `@contacts`. Once again using the `assigns()` method, we check that the collection (*assigned* to `@contacts`) is what we'd expect it to be with RSpec's `match_array` matcher. In this case, it's looking for a

single-item array containing the `smith` created within the example, but not `jones`. The second example makes sure that the view template `index.html.erb` is rendered, via `response`.

`match_array` looks for an array's contents, but not their order. If order matters, use the `eq` matcher instead.

The second context follows the same basic constructs; the only real difference is we're not passing a letter as a parameter to the method. As a result, in the first expectation *both* of the generated contacts are returned. Yes, I know there is some repetition here. Please work with it for now; typing it will help you learn the syntax. We'll clean this up soon, I promise.

new and *edit* are all that are left of the GET methods; let's add them now:

spec/controllers/contacts_controller_spec.rb

```
describe 'GET #new' do
  it "assigns a new Contact to @contact" do
    get :new
    expect(assigns(:contact)).to be_a_new(Contact)
  end

  it "renders the :new template" do
    get :new
    expect(response).to render_template :new
  end
end

describe 'GET #edit' do
  it "assigns the requested contact to @contact" do
    contact = create(:contact)
    get :edit, id: contact
    expect(assigns(:contact)).to eq contact
  end

  it "renders the :edit template" do
    contact = create(:contact)
    get :edit, id: contact
    expect(response).to render_template :edit
  end
end
```

Read through these examples–as you can see, once you know how to test one typical GET-based method, you can test most of them with a standard set of conventions.

Testing POST requests

Time to move on to our controller's *create* method, accessed via POST in our RESTful app. One key difference from the GET methods: Instead of the `:id` we passed to the GET methods, we need to pass the equivalent of `params[:contact]`–the contents of the form in which a user would enter a new contact. As mentioned earlier, we'll use Factory Girl's `attributes_for()` method to simulate the browser-to-server interaction. Here's the basic approach:

```
it "does something upon post#create" do
  post :create, contact: attributes_for(:contact)
end
```

With that in mind, here are some specs for the method in questions. First, with valid attributes:

spec/controllers/contacts_controller_spec.rb

```
1  describe "POST #create" do
2    before :each do
3      @phones = [
4        attributes_for(:phone),
5        attributes_for(:phone),
6        attributes_for(:phone)
7      ]
8    end
9
10   context "with valid attributes" do
11     it "saves the new contact in the database" do
12       expect{
13         post :create, contact: attributes_for(:contact,
14           phones_attributes: @phones)
15       }.to change(Contact, :count).by(1)
16     end
17
18     it "redirects to contacts#show" do
19       post :create, contact: attributes_for(:contact,
20         phones_attributes: @phones)
21       expect(response).to redirect_to contact_path(assigns[:contact])
22     end
23   end
```

And close out the block with invalid attributes:

spec/controllers/contacts_controller_spec.rb

```
  context "with invalid attributes" do
    it "does not save the new contact in the database" do
      expect{
        post :create,
          contact: attributes_for(:invalid_contact)
      }.to_not change(Contact, :count)
    end

    it "re-renders the :new template" do
      post :create,
        contact: attributes_for(:invalid_contact)
      expect(response).to render_template :new
    end
  end
end
```

There are a handful of things to take note of in this code:

First, check out the use of `context` blocks, as first introduced in chapter 3. Remember, although `describe` and `context` may be used interchangeably, it's considered best practice to use `context` when describing different *states*–in this case, one state with valid attributes, and one with invalid attributes. The examples using invalid attributes use the `:invalid_contact` factory we set up way back at the beginning of this chapter.

Second, look at the `before` hook at the beginning of the `describe` block. Given the validation requirement we included in our Contact model (that is, that an instance of Contact must have three phones associated with it in order to be valid), we've got to make sure to pass some phone attributes, too. This is one way to do it, by creating an array of three sets of phone attributes to pass into the POST request. Later we'll take a look at other, more efficient options. Ultimately, this may point to a code smell within the actual application, but let's work with it for now.

If you really want to get fancier with `attributes_for` and associations, check out custom strategies and custom callbacks[29], from the Factory Girl README.

Finally, take a look at the slight difference in how we're using `expect` in the first example. This time, we're passing the full HTTP request to `expect` in a block. This is slightly more complex than how we've been using `expect` so far. The HTTP request is passed in as a Proc, and the results are evaluated before and after, making it simple to determine whether the anticipated change happened–or in the case of this example, did *not* happen.

As usual, though, RSpec's readability shines here–*expect* this code to (or to not) do something. This one little example succinctly tests that an object is created and stored. Become familiar with this technique, as it'll be very useful in testing a variety of methods in controllers, models, and eventually at the integration level.

[29] https://github.com/thoughtbot/factory_girl/blob/master/GETTING_STARTED.md#custom-strategies

Testing PATCH requests

On to our controller's *update* method, where we need to check on a couple of things–first, that the attributes passed into the method get assigned to the model we want to update; and second, that the redirect works as we want. Let's take advantage of Rails 4.0's use of the HTTP verb PATCH.

Older versions of Rails will use PUT instead of PATCH.

spec/controllers/contacts_controller_spec.rb

```
describe 'PATCH #update' do
  before :each do
    @contact = create(:contact,
      firstname: 'Lawrence', lastname: 'Smith')
  end

  context "valid attributes" do
    it "locates the requested @contact" do
      patch :update, id: @contact, contact: attributes_for(:contact)
      expect(assigns(:contact)).to eq(@contact)
    end

    it "changes @contact's attributes" do
      patch :update, id: @contact,
        contact: attributes_for(:contact,
          firstname: "Larry", lastname: "Smith")
      @contact.reload
      expect(@contact.firstname).to eq("Larry")
      expect(@contact.lastname).to eq("Smith")
    end

    it "redirects to the updated contact" do
      patch :update, id: @contact, contact: attributes_for(:contact)
      expect(response).to redirect_to @contact
    end
  end

  # ...
end
```

Then, as we did in the previous POST examples, we need to test that those things *don't* happen if invalid attributes are passed through the params:

spec/controllers/contacts_controller_spec.rb

```
describe 'PATCH #update' do
  # ...

  context "with invalid attributes" do
    it "does not change the contact's attributes" do
      patch :update, id: @contact,
        contact: attributes_for(:contact,
          firstname: "Larry", lastname: nil)
      @contact.reload
      expect(@contact.firstname).to_not eq("Larry")
      expect(@contact.lastname).to eq("Smith")
    end

    it "re-renders the edit template" do
      patch :update, id: @contact,
        contact: attributes_for(:invalid_contact)
      expect(response).to render_template :edit
    end
  end
end
```

Points of interest:

- Since we're updating an existing Contact, we need to persist something first. We take care of that in the `before` hook, making sure to assign the persisted Contact to `@contact` to access it later. (Again, we'll look at more appropriate ways to do this in later chapters.)
- The two examples that verify whether or not an object's attributes are actually changed by the *update* method–we can't use the `expect{}` Proc here. Instead, we have to call `reload` on `@contact` to check that our updates are actually persisted. Otherwise, these examples follow a similar pattern to the one we used in the POST-related specs.

Testing DELETE requests

After all that, testing the *destroy* method is relatively straightforward:

spec/controllers/contacts_controller_spec.rb

```
describe 'DELETE #destroy' do
  before :each do
    @contact = create(:contact)
  end

  it "deletes the contact" do
    expect{
      delete :destroy, id: @contact
    }.to change(Contact,:count).by(-1)
  end

  it "redirects to contacts#index" do
    delete :destroy, id: @contact
    expect(response).to redirect_to contacts_url
  end
end
```

By now you should be able to correctly guess what everything's doing. The first expectation checks to see if the `destroy` method in the controller actually deleted the object (using the now-familiar `expect{}` Proc); the second expectation confirms that the user is redirected back to the index upon success.

Testing non-CRUD methods

Testing a controller's other methods isn't much different from testing the standard, out-of-the-box RESTful resources Rails gives us. Let's use the hypothetical example of a `hide_contact` method on ContactsController, which provides administrators with a convenient means of hiding contacts from view without deleting them (I'll leave it to you to implement this functionality, if you'd like).

We could test this at the controller level with something like

```
describe "PATCH hide_contact" do
  before :each do
    @contact = create(:contact)
  end

  it "marks the contact as hidden" do
    patch :hide_contact, id: @contact
    expect(@contact.reload.hidden?).to be_true
  end

  it "redirects to contacts#index" do
```

```
12      patch :hide_contact, id: @contact
13      expect(response).to redirect_to contacts_url
14    end
15  end
```

See what we're doing? We're using the PATCH method–since we're editing an existing contact–along with `:hide_contact` to indicate the controller method to access. Everything else works similarly to testing the update method, with the exception that we're not passing a hash of user-entered attributes–in this example, the `hidden?` boolean is set server-side.

`expect(@contact.reload.hidden?).to be_true` is a good candidate for a custom matcher. We'll visit this concept in chapter 7.

If your method uses one of the other HTTP request methods, just follow along with its respective CRUD-based approach to test it.

Testing nested routes

If your application uses *nested routes*–that is, a route that looks something like `/contacts/34/phones/22`, you'll need to provide your examples with a little more information.

See Rails Routing from the Outside In[30] for an excellent overview of nested routes.

In another hypothetical example, let's say we implemented phones with nested routes instead of nested attributes. This means that, instead of inputting each phone's attributes within its associated contact's form, we need a separate controller/views combination to gather and process phone data. The route configuration in *config/routes.rb* would look something like:

config/routes.rb

```
1  resources :contacts do
2    resources :phones
3  end
```

If you were to look at the routes for the app (with `rake routes`, via the command line) you'd see that the path to PhoneController's `:show` method translates to `/contacts/:contact_id/phones/:id`–so we need to pass in a phone's `:id` *and* a `:contact_id` for its parent contact. Here's how that would look in a spec:

[30] http://guides.rubyonrails.org/routing.html#nested-resources

```
describe 'GET #show' do
  it "renders the :show template for the phone" do
    contact = create(:contact)
    phone = create(:phone, contact: contact)
    get :show, id: phone, contact_id: contact.id
    expect(response).to render_template :show
  end
end
```

The key thing to remember is you need to pass the parent route to the server in the form of `:parent_id`–in this case, `:contact_id`. The controller will handle things from there, as directed by your *routes.rb* file. This same basic technique applies to any method in a nested controller.

Testing non-HTML controller output

So far we've just been testing a controller method's HTML output. Of course, Rails lets us send multiple data types from a single controller method in addition to–or instead of–HTML.

Continuing with our hypothetical examples, let's say we need to export contacts to a CSV file. If you're already returning content in a non-HTML format in your own applications, you probably know how to override the HTML default in a given method's route:

```
link_to 'Export', contacts_path(format: :csv)
```

This would assume a controller method along these lines:

```
def index
  @contacts = Contact.all

  respond_to do |format|
    format.html # index.html.erb
    format.csv do
      send_data Contact.to_csv(@contacts),
        type: 'text/csv; charset=iso-8859-1; header=present',
          disposition: 'attachment; filename=contacts.csv'
    end
  end
end
```

A simple means of testing this, then, is to verify the data type:

```
describe 'CSV output' do
  it "returns a CSV file" do
    get :index, format: :csv
    expect(response.headers['Content-Type']).to have_content 'text/csv'
  end

  it 'returns content' do
    create(:contact,
      firstname: 'Aaron',
      lastname: 'Sumner',
      email: 'aaron@sample.com')
    get :index, format: :csv
    expect(response.body).to have_content 'Aaron Sumner,aaron@sample.com'
  end
end
```

The `have_content` matcher shown here comes from Capybara, which we'll cover with more depth in chapter 8.

This will verify that the controller is returning the CSV data with the proper content type. However, given the structure we're using to actually *generate* CSV content–that is, with a class method on *Contact*, testing that functionality at the model level (as opposed to the controller layer) is perhaps the ideal way to go:

```
it "returns comma separated values" do
  create(:contact,
    firstname: 'Aaron',
    lastname: 'Sumner',
    email: 'aaron@sample.com')
  expect(Contact.to_csv).to match /Aaron Sumner,aaron@sample.com/
end
```

Note the use of RSpec's `match` matcher here, used whenever a regular expression is being compared to the actual results.

To see the general approach I've used to generate CSV-formatted data, see Railscasts episode 362, Exporting to CSV and Excel[31].

If you're using Rails to serve up an API, it's also possible to test JSON or XML output with relative ease at the controller level:

[31] http://railscasts.com/episodes/362-exporting-csv-and-excel

```
it "returns JSON-formatted content" do
  contact = create(:contact)
  get :index, format: :json
  expect(response.body).to have_content contact.to_json
end
```

The key when using this approach is to keep your code nice and modular, so you can directly test the class in question (in this case, a controller class). From there you might want to explore Rack::Test[32]. If you're using the API to render a JavaScript-based interface within your own application, then using Capybara and feature specs could be a useful addition to your test suite (we'll cover feature specs and Capybara in chapter 8).

A complete example is beyond the scope of this book, but a good resource for testing JSON APIs in Rails is *Rails 3 in Action*[33] by Ryan Bigg and Yehuda Katz. You may also want to explore client-side test libraries like Jasmine[34]. Noel Rappin's *Master Space and Time with JavaScript*[35] series is a good resource to learn about testing at this level.

Summary

In a nutshell, that's how you test your application's controllers. The key is to break down what you need to test, and then incrementally build those tests until you've got your functionality covered.

Unfortunately, controller specs aren't always this straightforward. Often you'll need to contend with user logins, additional, non-scaffolded code, or models with particular validation requirements. That's what we'll cover next.

Exercises

- Thinking back to chapter 4, how would you test the `:invalid_contact` factory?
- The astute reader may spot a design flaw in the example controller's *index* method. Can you see what it is? Can you refactor the code to reduce logic in the controller? See the next exercise for a hint.
- If you're noticing you've got to do quite a bit of setup just to test a single controller method, it could be a sign that your controller needs refactoring. Perhaps there's code contained within the controller method that would be better-suited in a model or helper method. Take opportunities to clean up your code as they present themselves–move the offending code to a model, spec it there as needed, and simplify your controller spec–which should still pass.

[32] http://rubygems.org/gems/rack-test
[33] http://manning.com/katz/
[34] https://github.com/pivotal/jasmine
[35] https://noelrappin.dpdcart.com/

6. Advanced controller specs

With the basics of controller testing out of the way, it's time to return to some real code and look at how RSpec helps make sure our application's controllers are doing what we expect them to do. This time, though, we'll build onto the vanilla CRUD specs by accounting for our application's authentication and authorization layers. In a bit more detail:

- We'll start by setting up a more complicated spec.
- Next we'll cover testing authentication, or login requirements, through the controller.
- We'll follow that by testing authorization, or roles; also through the controller.
- We'll also look at a technique for making sure controller specs are properly handling any additional setup requirements your application might have.

Check out the *06_advanced_controllers* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 06_advanced_controllers origin/06_advanced_controllers
```

See chapter 1 for additional details.

Getting ready

In the previous chapter, we commented out the `before_action` to authenticate ContactsController. Uncomment the line to re-enable authentication. Run RSpec and look at how many specs are now breaking!

```
Finished in 1.04 seconds
32 examples, 13 failures
```

We need a way to mimic the authorization process in the controller specs to continue. In particular, we've got to contend with whether a user is logged in or not, and the logged-in user's role (as a reminder, you must be a user to add or edit, and you must be an administrative user to add other users). We'll use a basic mechanism in our application controller to handle this authorization layer in the app itself, and test the settings at the controller level.

Testing the admin and user roles

We're going to take a different approach to walking through the controller spec this time. We'll go through each possible role–guest, user, and administrator. Let's start by fixing those currently-failing specs. In our application, users and administrators (users with the `:admin` boolean enabled) have identical permissions for contacts: Any user who has signed in with a valid account can create, edit, and delete any contact.

First, let's establish a new factory for users:

spec/factories/users.rb

```
require 'faker'

FactoryGirl.define do
  factory :user do
    email { Faker::Internet.email }
    password 'secret'
    password_confirmation 'secret'

    factory :admin do
      admin true
    end
  end
end
```

This will allow us to quickly create a new user object with `create(:user)` (or `FactoryGirl.create(:user)` when not using shorthand); it also creates a child factory called `admin` for creating users with the administrator role, by setting the `admin` boolean to true.

Now, back to the controller spec, let's use the factory to test administrator access. Pay close attention to the first few lines:

spec/controllers/contacts_controller_spec.rb

```
describe "administrator access" do
  before :each do
    user = create(:admin)
    session[:user_id] = user.id
  end

  describe 'GET #index' do
    it "populates an array of contacts" do
      get :index
      expect(assigns(:contacts)).to match_array [@contact]
    end

    it "renders the :index template" do
      get :index
      expect(response).to render_template :index
    end
  end

  describe 'GET #show' do
    it "assigns the requested contact to @contact" do
      get :show, id: @contact
      expect(assigns(:contact)).to eq @contact
```

```
    end

    it "renders the :show template" do
      get :show, id: @contact
      expect(response).to render_template :show
    end
  end

  # and so on ...
end
```

What's going on here? It's pretty simple, really: I start by wrapping all of my existing examples in the spec inside a new `describe` block, then add a `before` block inside it to mimic logging in as an administrator. This is done by first instantiating an administrator object with the new `:admin` factory, then by assigning it to the session value directly.

In this case, that's all there is to it. With a valid login being simulated, our controller specs are passing again.

For the sake of brevity, I'm not going to include a full non-administrator's permission specs. Review the sample code for this chapter for a complete implementation–aside from `before :each` block the expectations are exactly the same.

spec/controllers/contacts_controller_spec.rb

```
describe "user access" do
  before :each do
    user = create(:user)
    session[:user_id] = user.id
  end

  # specs are the same as administrator
```

Yes, this is a lot of redundant code. Don't worry, RSpec has a feature to deal with this. We'll get to it in the next chapter. For now, just focus on the different use cases we need to test.

Testing the guest role

It may often be easy to overlook the guest role–that is, a user who's not logged in. However, in a public-facing application like ours, that may be the most common role! Let's add it to the spec. Unlike specs for our other roles, we'll need to make some changes to these–they're not direct copy-and-paste jobs, but they are pretty easy to write:

spec/controllers/contacts_controller_spec.rb

```
describe "guest access" do
  # GET #index and GET #show examples are the same as those for
  # administrators and users

  describe 'GET #new' do
    it "requires login" do
      get :new
      expect(response).to redirect_to login_url
    end
  end

  describe 'GET #edit' do
    it "requires login" do
      contact = create(:contact)
      get :edit, id: contact
      expect(response).to redirect_to login_url
    end
  end

  describe "POST #create" do
    it "requires login" do
      post :create, id: create(:contact),
        contact: attributes_for(:contact)
      expect(response).to redirect_to login_url
    end
  end

  describe 'PUT #update' do
    it "requires login" do
      put :update, id: create(:contact),
        contact: attributes_for(:contact)
      expect(response).to redirect_to login_url
    end
  end

  describe 'DELETE #destroy' do
    it "requires login" do
      delete :destroy, id: create(:contact)
      expect(response).to redirect_to login_url
    end
  end
end
```

Nothing new until we hit the *new* block–remember, the first method the controller's `before_-`

`action` requires login to access. This time, we need to make sure guests *can't* do the things in the controller method–instead, they should be redirected to the `login_url`, at which point they will be asked to sign in. As you can see, we can use this technique on *any* method that requires login.

Running the specs again, they should now pass. As an experiment, comment out the `before_action :authenticate` line in the controller again, and run the specs to see what happens. You can also change `expect(response).to redirect_to login_url` to `expect(response).to_not redirect_to login_url`, or change `login_url` to a different path.

It's a good idea to intentionally break things like this, to help reduce the chance a false positive in a test gets past you.

Testing a given role's authorization

Finally, we'll need to consider a different controller to see how to spec a given user's authorization–that is, what he or she is allowed to do upon successful login. In the sample application, only administrators may add new users. Regular users–those without the `:admin` boolean switched on–should be denied access.

The approach is basically the same one we've followed so far: Set up a user to simulate in a `before :each` block, simulate the login by assigning the `user_id` to a session variable in the `before :each` block, then write the specs. This time, though, instead of redirecting to the login form, users should be redirected back to the application's root URL (this is the behavior defined in `app/controllers/application_controller.rb`). Here are some specs for this scenario:

spec/controllers/users_controller_spec.rb

```
1  describe 'user access' do
2    before :each do
3      @user = create(:user)
4      session[:user_id] = @user.id
5    end
6
7    describe 'GET #index' do
8      it "collects users into @users" do
9        user = create(:user)
10       get :index
11       expect(assigns(:users)).to match_array [@user,user]
12     end
13
14     it "renders the :index template" do
15       get :index
16       expect(response).to render_template :index
17     end
18   end
```

```
19
20    it "GET #new denies access" do
21      get :new
22      expect(response).to redirect_to root_url
23    end
24
25    it "POST#create denies access" do
26      post :create, user: attributes_for(:user)
27      expect(response).to redirect_to root_url
28    end
29  end
```

Summary

We've covered an awful lot in the past couple of chapters–but the fact is, you can test an awful lot of your application's functionality by applying good test coverage at the controller level.

As I shared in the previous chapter, I don't always test my own controllers with such thoroughness. I tend to leverage controller specs on a case-by-case basis (typically, for non-boilerplate code). That said, as you can see from RSpec's generated examples, there are several things you can–and should–test at the controller level.

And with thoroughly tested controllers, you're well on your way to thorough test coverage in your application as a whole. By now you should be getting a handle on good practices and techniques for the practical use of RSpec, Factory Girl, and other helpers to make your tests and code more reliable.

We can still do better, though–let's go through this spec one more time and clean it up through helper methods and shared examples.

Exercise

For a given controller in your application, sketch a table of which methods should be accessible to which users. For example, say I have a blogging application for premium content–users must become members to access content, but can get a feel for what they're missing by seeing a list of titles. Actual users have different levels of access based on their respective roles. The hypothetical app's Posts controller might have the following permissions:

Role	Index	Show	Create	Update	Destroy
Admin	Full	Full	Full	Full	Full
Editor	Full	Full	Full	Full	Full
Author	Full	Full	Full	Full	None
Member	Full	Full	None	None	None
Guest	Full	None	None	None	None

Use this table to help figure out the various scenarios that need to be tested. In this example I

merged *new* and *create* into one column (since it doesn't make much sense to render the *new* form if it can't be used to create anything), as well as *edit* and *update*, while splitting *index* and *show*. How would these compare to your application's authentication and authorization requirements? What would you need to change?

7. Controller spec cleanup

If you've been applying what you've learned so far to your own code, you're well on your way to a solid test suite. However, in the last chapter we introduced a lot of repetition–and potentially brittle tests. What would happen, say, if instead of redirecting unauthorized requests to `root_path`, we created a specific `denied_path` route? We've have a lot of individual specs to clean up.

Just as you would your application code, you should take opportunities to clean up your specs. In this chapter we'll look at three ways to reduce redundancy and brittleness, without sacrificing readability:

- To start, we'll share examples across multiple `describe` and `context` blocks.
- Next we'll reduce more repetition with helper macros.
- We'll finish up by creating custom RSpec matchers.

Check out the *07_controller_cleanup* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 07_controller_cleanup origin/07_controller_cleanup
```

See chapter 1 for additional details.

Shared examples

Way back in chapter 1, when discussing my general approach to testing, I said a readable spec is ultimately more important than a 100 percent DRY spec. I stand by that–but looking at `contacts_controller_spec.rb`, something's got to give. As it stands right now, we've got many examples included twice (once for administrators; once for regular users), and some examples are included *thrice*–guests, admins, and regular users may all access the `:index` and `:show` methods. That's a lot of code, and it jeopardizes readability and long-term maintainability.

RSpec gives us a nice way to clean up this replication with *shared examples*. Setting up a shared example is pretty simple–first, create a block for the examples as follows:

spec/controllers/contacts_controller_spec.rb

```
shared_examples("public access to contacts") do
  describe 'GET #index' do
    it "populates an array of contacts" do
      get :index
      expect(assigns(:contacts)).to match_array [@contact]
    end

    it "renders the :index template" do
      get :index
      expect(response).to render_template :index
    end
  end

  describe 'GET #show' do
    it "assigns the requested contact to @contact" do
      get :show, id: @contact
      expect(assigns(:contact)).to eq @contact
    end

    it "renders the :show template" do
      get :show, id: @contact
      expect(response).to render_template :show
    end
  end
end
```

Then include them in any `describe` or `context` block in which you'd like to use the examples, like this (actual code removed for clarity here; refer to the file in the sample source for context):

spec/controllers/contacts_controller_spec.rb

```
describe "guest access" do
  it_behaves_like "public access to contacts"

  # rest of specs for guest access ...
end
```

As a result of this exercise, our `contacts_controller_spec.rb` is much cleaner, as you can see in this outline:

spec/controllers/contacts_controller_spec.rb

```
require 'spec_helper'

describe ContactsController do
  shared_examples("public access to contacts") do
    describe 'GET #index' do
      it "populates an array of contacts"
      it "renders the :index template"
    end

    describe 'GET #show' do
      it "assigns the requested contact to @contact"
      it "renders the :show template"
    end
  end

  shared_examples("full access to contacts") do
    describe 'GET #new' do
      it "assigns a new Contact to @contact"
      it "assigns a home, office, and mobile phone to the new contact"
      it "renders the :new template"
    end

    describe 'GET #edit' do
      it "assigns the requested contact to @contact"
      it "renders the :edit template"
    end

    describe "POST #create" do
      context "with valid attributes" do
        it "creates a new contact"
        it "redirects to the new contact"
      end

      context "with invalid attributes" do
        it "does not save the new contact"
        it "re-renders the new method"
      end
    end

    describe 'PATCH #update' do
      context "valid attributes" do
        it "located the requested @contact"
        it "changes @contact's attributes"
        it "redirects to the updated contact"
```

```
45        end
46
47        context "invalid attributes" do
48          it "locates the requested @contact"
49          it "does not change @contact's attributes"
50          it "re-renders the edit method"
51        end
52      end
53
54      describe 'DELETE destroy' do
55        it "deletes the contact"
56        it "redirects to contacts#index"
57      end
58    end
59
60    describe "admin access to contacts" do
61      before :each do
62        set_user_session(create(:admin))
63      end
64
65      it_behaves_like "public access to contacts"
66      it_behaves_like "full access to contacts"
67    end
68
69    describe "user access to contacts" do
70      before :each do
71        set_user_session(create(:user))
72      end
73
74      it_behaves_like "public access to contacts"
75      it_behaves_like "full access to contacts"
76    end
77
78    describe "guest access to contacts" do
79      it_behaves_like "public access to contacts"
80
81      describe 'GET #new' do
82        it "requires login"
83      end
84
85      describe "POST #create" do
86        it "requires login"
87      end
88
89      describe 'PATCH #update' do
90        it "requires login"
```

```
91      end
92
93      describe 'DELETE #destroy' do
94        it "requires login"
95      end
96    end
97  end
```

And when we run `bundle exec rspec controllers/contacts_controller_spec.rb`, the documentation output is just as readable:

```
1  ContactsController
2    admin access to contacts
3      behaves like public access to contacts
4        GET #index
5          populates an array of contacts
6          renders the :index template
7        GET #show
8          assigns the requested contact to @contact
9          renders the :show template
10     behaves like full access to contacts
11       GET #new
12         assigns a new Contact to @contact
13         assigns a home, office, and mobile phone to the new contact
14         renders the :new template
15       GET #edit
16         assigns the requested contact to @contact
17         renders the :edit template
18       POST #create
19         with valid attributes
20           creates a new contact
21           redirects to the new contact
22         with invalid attributes
23           does not save the new contact
24           re-renders the new method
25       PATCH #update
26         valid attributes
27           located the requested @contact
28           changes @contact's attributes
29           redirects to the updated contact
30         invalid attributes
31           locates the requested @contact
32           does not change @contact's attributes
33           re-renders the edit method
34       DELETE destroy
```

```
35          deletes the contact
36          redirects to contacts#index
37    user access to contacts
38      behaves like public access to contacts
39        GET #index
40          populates an array of contacts
41          renders the :index template
42        GET #show
43          assigns the requested contact to @contact
44          renders the :show template
45      behaves like full access to contacts
46        GET #new
47          assigns a new Contact to @contact
48          assigns a home, office, and mobile phone to the new contact
49          renders the :new template
50        GET #edit
51          assigns the requested contact to @contact
52          renders the :edit template
53        POST #create
54          with valid attributes
55            creates a new contact
56            redirects to the new contact
57          with invalid attributes
58            does not save the new contact
59            re-renders the new method
60        PATCH #update
61          valid attributes
62            located the requested @contact
63            changes @contact's attributes
64            redirects to the updated contact
65          invalid attributes
66            locates the requested @contact
67            does not change @contact's attributes
68            re-renders the edit method
69        DELETE destroy
70          deletes the contact
71          redirects to contacts#index
72    guest access to contacts
73      behaves like public access to contacts
74        GET #index
75          populates an array of contacts
76          renders the :index template
77        GET #show
78          assigns the requested contact to @contact
79          renders the :show template
80      GET #new
```

```
      requires login
    POST #create
      requires login
    PATCH #update
      requires login
    DELETE #destroy
      requires login
```

Creating helper macros

Now let's turn our attention to another bit of code we've used several times in our controllers. Whenever we're testing what a logged-in user can or can't do, we simulate that login by setting a session value to a factory-generated user's `:id`. Let's move this functionality to a *macro* in RSpec. Macros are an easy way to create methods which may be used across your entire test suite. Macros conventionally go into the *spec/support* directory as a module to be included in RSpec's configuration.

First, here's a macro for setting that session variable:

spec/support/login_macros.rb

```
module LoginMacros
  def set_user_session(user)
    session[:user_id] = user.id
  end
end
```

Just a simple Ruby module and method–it accepts a `user` object and assigns `session[:user_id]` to that user's own `:id`.

Before we can use this new helper in our specs, we've got to let RSpec know where to find it. Inside the `RSpec.configure` block in *spec/spec_helper.rb*, we'll add the line `config.include LoginMacros` as shown below:

spec/spec_helper.rb

```
Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}

RSpec.configure do |config|
  # other RSpec configuration omitted ...

  config.include LoginMacros
end
```

Authentication options like Devise offer similar functionality. If you're using such a solution in your project, refer to its documentation for instructions on incorporating it into your test suite.

With that in place, let's apply it to a controller spec. In a `before` block, we'll create a new admin user, then set the session to that user–all in a single line:

spec/controllers/contacts_controller_spec.rb

```
describe "admin access" do
  before :each do
    set_user_session create(:admin)
  end

  it_behaves_like "public access to contacts"
  it_behaves_like "full access to contacts"
end
```

It might seem silly to create a whole separate helper method for just one line of code, but in reality it could come to be that we change up our whole authentication system, and need to simulate login in a different fashion. By simulating login in this manner, we just have to make the change in one place.

When we get to integration testing in the next chapter, this technique might help us reuse *several* lines of code, as we simulate each step of a user login.

Using custom RSpec matchers

So far we've gotten a lot of mileage out of RSpec's built-in matchers–and truth be told, you could probably test an entire application without ever straying from the standards. (I know; I have.) However, just as we reviewed when building some helper macros in the last section, adding a few custom matchers for your application can boost your test suite's long-term reliability. In the case of our address book, what if we changed the route for the login form or where we direct users who try to access more than they're allowed? As it is we'd have a lot of examples to switch out to the new route–or we could set up a custom matcher and just change the route in one place. If you store custom matchers in *spec/support/matchers*, one matcher per file, the default RSpec configuration will automatically pick them up for use in your specs.

Here's an example:

spec/support/matchers/require_login.rb

```
RSpec::Matchers.define :require_login do |expected|
  match do |actual|
    expect(actual).to \
      redirect_to Rails.application.routes.url_helpers.login_path
  end

  failure_message_for_should do |actual|
    "expected to require login to access the method"
  end
```

```
10
11    failure_message_for_should_not do |actual|
12      "expected not to require login to access the method"
13    end
14
15    description do
16      "redirect to the login form"
17    end
18  end
```

Let's take a quick tour of this code: the `match` block is what we *expect* to happen–essentially, replacing the code after `expect(something).to` in a given spec. Note that RSpec doesn't load Rails' `UrlHelpers` library, so we'll give the matcher a little help by calling its full path. We check to see if the *actual* value we're passing to the matcher (in this case, `response`) does what we expect it to (redirect to our login form). If it does, the matcher reports success.

Next we can provide some helpful messages to return if an example using the matcher doesn't pass–the first one is a message for when we expect the matcher to return true, and the second for when it should return false. In other words, a single matcher covers both `expect(:foo).to` and `expect(:foo).to_not`–no need to write two matchers.

Even though RSpec's syntax for expectations has moved from `should` to `expect`, custom matchers still use `failure_message_for_should` and `failure_message_for_should_not`. This will likely change in RSpec 3.

Now, replacing the matcher in our examples is easy:

spec/controllers/contacts_controller_spec.rb

```
1  describe 'GET #new' do
2    it "requires login" do
3      get :new
4      expect(response).to require_login
5    end
6  end
```

This is just one example of what we can do with custom matchers. Coming up with a fancier example at this point in our application would be contrived (and possibly confusing), so I suggest reviewing the sample matchers in RSpec's documentation[36].

Summary

If left unkempt, controller specs can sprawl out of control pretty quickly–but with a little management (and some help from RSpec's useful support methods) you can keep things in check

[36] https://www.relishapp.com/rspec/rspec-expectations/v/2-3/docs/custom-matchers

for solid long-term maintenance. Just as you shouldn't ignore control specs, please also don't ignore your responsibility of keeping these specs pruned and tidy. Your future self will thank you for it.

We've spent a lot of time testing controllers. As I said way back at the beginning of chapter 5, testing at this level is an economical way to build confidence in a large swath of your code base–and is good practice for testing at other levels, since many of the same concepts can apply to other levels of testing. By keeping these tests clean and readable, you'll be sure to make that confidence last for the lifetime of your application.

One more level of testing to go: Integration. The work we've done so far has given us comfort in our application's building blocks–next let's make sure they fit nicely together into a cohesive structure.

Exercises

- Examine your own test suite and look for places to tidy up. Controller specs are a prime target, but check your model specs, too. What are the best methods for cleaning up each section–shared examples? A custom matcher? A helper macro? Update your specs as needed, making sure they continue to pass along the way.
- Back in chapter 5, I mentioned that `expect(@contact.reload.hidden?).to be_true` could be streamlined with a custom matcher. What would that look like?

8. Feature specs

So far we've added a good amount of test coverage to our contacts manager. We got RSpec installed and configured, set up some unit tests on models and controllers, and used factories to generate test data. Now it's time to put everything together for integration testing–in other words, making sure those models and controllers all play nicely with other models and controllers in the application. These tests are called *feature specs* in RSpec. Once you get the hang of them, they can be used to test a wide range of functionality within a Rails application. They may also be used to replicate bug reports from your application's users.

The good news is you know almost everything you need to know to write solid feature specs–they follow a similar structure you've been using in models and controllers, and you can use Factory Girl to generate test data for them. The star of the show, though, is *Capybara*, an extremely useful Ruby library to help define steps of a feature spec and simulate real-world use of your application.

In this chapter, we'll look at the nuts and bolts of an RSpec feature spec:

- We'll start with some thoughts on when and why feature specs make sense versus other options.
- Next we'll cover a few additional dependencies to aid in integration testing.
- Then we'll look at a basic feature spec.
- After that we'll tackle a slightly more advanced approach, with JavaScript requirements incorporated.
- Finally, we'll close with some discussion on best practices for feature specs.

Check out the *08_features* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 08_features origin/08_features
```

See chapter 1 for additional details.

Why feature specs?

We just spent a *lot* of time going over controller testing. After all that, why are we doing another layer of tests? Because controller tests are relatively simple *unit tests* and, while they test an important component of your software, they are only testing a small part of an application. A feature spec covers more ground, and represents how actual users will interact with your code.

What about Cucumber?

Cucumber[37] is a popular alternative to the type of tests we'll be working on in this chapter. To be honest, I've run hot and cold with it for a few years now. It's definitely got its uses–but it's also got a lot of overhead and, unless you know how to use it correctly, can lead to brittle and ultimately useless tests. I can understand wanting to use Cucumber if you're working directly *with* a non-programmer product owner who doesn't want to look at a lot of code, but from my experience Capybara's DSL is understandable enough that non-programmers can still read through a feature spec and understand what's going on. And if you're *not* working with a non-programmer, then the extra overhead incumbent with Cucumber may not be worth the effort.

Of course, Cucumber does have its ardent supporters. It's a staple in many development shops, so you'll probably need to become familiar with it eventually, too. The good news is, if you do want or need to use Cucumber down the road, understanding how Capybara and RSpec work at the feature spec level will make things easier to understand.

If you do go the Cucumber route, be mindful of any tutorial that existed prior to December, 2011. That's when it was revealed that Cucumber's *web_steps.rb* file–the helpers that let you add steps like `When I fill in "Email" with "aaron@everydayrails.com"`–were deemed "training wheels."[38] Post-December 2011, it's recommended that you make your Cucumber scenarios more direct, and leave the heavy lifting to custom step definitions using Capybara.

Additional dependencies

Way back in chapter 2, we included Capybara[39], DatabaseCleaner[40], and Launchy[41] in our Gemfile's `test` group. If you haven't added them yet, do so now–we're finally going to put them to use.

Gemfile

```
group :test do
  gem "faker", "~> 1.1.2"
  gem "capybara", "~> 2.1.0"
  gem "database_cleaner", "~> 1.0.1"
  gem "launchy", "~> 2.3.0"
end
```

DatabaseCleaner will need some configuration regardless of your installed version of RSpec, but let's look at a simple spec that doesn't require DatabaseCleaner first.

[37] http://cukes.info
[38] http://aslakhellesoy.com/post/11055981222/the-training-wheels-came-off
[39] https://github.com/jnicklas/capybara
[40] https://github.com/bmabey/database_cleaner
[41] http://rubygems.org/gems/launchy

A basic feature spec

Capybara lets you simulate how a user would interact with your application through a web browser, using a series of easy-to-understand methods like `click_link`, `fill_in`, and `visit`. What these methods let you do, then, is describe a test scenario for your app. Can you guess what this feature spec does?

spec/features/users_spec.rb

```
require 'spec_helper'

feature 'User management' do
  scenario "adds a new user" do
    admin = create(:admin)

    visit root_path
    click_link 'Log In'
    fill_in 'Email', with: admin.email
    fill_in 'Password', with: admin.password
    click_button 'Log In'

    visit root_path
    expect{
      click_link 'Users'
      click_link 'New User'
      fill_in 'Email', with: 'newuser@example.com'
      find('#password').fill_in 'Password', with: 'secret123'
      find('#password_confirmation').fill_in 'Password confirmation',
        with: 'secret123'
      click_button 'Create User'
    }.to change(User, :count).by(1)
    expect(current_path).to eq users_path
    expect(page).to have_content 'New user created'
    within 'h1' do
      expect(page).to have_content 'Users'
    end
    expect(page).to have_content 'newuser@example.com'
  end
end
```

Walking through the steps of this spec, you should be able to see that the spec first creates a new administrator (a user who can create other users), then uses the login form to sign in as that administrator and create a new user *using the same web form our application's administrators would use*. This is an important distinction between feature specs and controller specs–in controller specs, we bypass the user interface and send parameters directly to the controller method (which, in this case, would be *multiple* controllers and actions–`contacts#index`, `sessions#new`,

`users#new`, and `users#create`). However, the results should be the same–a new user is created, the application redirects to a listing of all users, a flash message is rendered to let us know the process was successful, and the new user is listed on the page.

You may also recognize some techniques from previous chapters–`feature` is used in place of `describe` to structure the spec, `scenario` describes a given example in place of `it`, and the `expect{}` Proc we checked out in chapter 5 plays the same role here–we *expect* that certain things will change when a user completes the scripted actions when interacting with the site.

See the use of `find('#password')` and `find('#password_confirmation)` here? As you may guess, this method *finds* elements on the rendered page, using whatever you pass into it as an argument (not to be confused with ActiveRecord's own `find` method). In this case it's finding by CSS–`<div>` elements by their `ids`. We could also find elements by XPath location, or just plain text as shown for `click_link 'Users'`, `fill_in 'Email'`, and so on. However, a spec will fail if a match is ambiguous–in other words, if I'd tried the following:

```
fill_in 'Password', with: 'secret'
fill_in 'Password confirmation', with: 'secret'
```

Capybara would have returned an *Ambiguous match* error. If you receive such an error, fire up the view file rendering your HTML and look for alternative ways to locate the field you want to manipulate. (Prior to Capybara 2.0, such use of `fill_in` wouldn't have resulted in this error.)

If possible, I prefer to stick with plain text matchers, then CSS; if neither of those matches exactly what I want I'll defer to XPath-based matchers. Refer to Capybara's README file for more information.

Following `expect{}`, we run a series of tests to make sure the resulting view is displayed in a way we'd expect, using Capybara's not-quite-plain-English-but-still-easy-to-follow style. Check out, too, the `within` block used to specify *where* to look on a page for specific content– in this case, within the `<h1>` tag in the `index` view for users. This is an alternative to the `find()` approach used to locate the password and password confirmation fields. You can get pretty fancy with this if you'd like–more on that in just a moment.

One final thing to point out here: Within feature specs, it's perfectly reasonable to have multiple expectations in a given example or scenario. Feature specs typically have much greater overhead than the smaller examples we've written so far models and controllers, and as such can take a lot longer to set up and run. You may also add expectations mid-test. For example, in the previous spec I may want to verify that the user is notified of successful login via a flash message–though in reality, such an expectation might be more appropriate in a feature spec dedicated to nuances of our application's login mechanism.

From requests to features

In November, 2012, Capybara 2.0 introduced a few changes to the DSL, including the aforementioned use of the term `feature` instead of `request`. Request specs still have a place, but are now intended to test any public API your application might serve.

In addition to moving the location of these specs, Capybara 2.0 introduced a few aliases to help feature specs feel a little more like acceptance tests written in other frameworks (read: Cucumber). These aliases–namely, the aforementioned `feature` and `scenario`–are exclusive to feature specs. Other aliases include `background` for `before` and `given` for `let` (which we'll cover in chapter 9). A couple of caveats: While tests at other levels of your app can nest `describe` blocks within other `describe` blocks, you can't nest `feature` blocks. You'll also likely encounter errors if you mix and match syntax. You can have multiple features per file, and multiple scenarios per feature, just as you can have multiple `it` blocks within a `describe`.

Strictly speaking, you *can* use `describe` and `it` in your feature specs, but for best results, I recommend using the new Capybara DSL. That's how we'll write our examples for our address book application moving forward.

Adding feature specs

The quickest way to add a new feature spec to your application is to create a new file inside `spec/features`, beginning with the following template:

```
1  require 'spec_helper'
2
3  feature 'my feature' do
4    background do
5      # add setup details
6    end
7
8    scenario 'my first test' do
9      # write the example!
10   end
11 end
```

As of this writing, if using Rails' scaffold generator to create models and their associated controllers, views, migrations, and specs, the corresponding feature spec will be added to `spec/requests`. Delete it, or move it to `spec/features` and edit. You can also make sure the scaffold generator doesn't create these files for you by making sure `request_specs: false` is included in your *application.rb* file's RSpec generator configuration.

Debugging feature specs

I've already mentioned that it's typical to see a given scenario in a feature have multiple expectations. However, that can sometimes lead you to wonder why a scenario might be failing at a certain point. For the most part, you can use the same tools you use to debug any Ruby application within RSpec–but one of the easiest to use is *Launchy*. Launchy does just one thing

when called: It saves the feature spec's current HTML to a temporary file and renders it in your default browser.

To use Launchy in a spec, add the following line anywhere you'd like to see the results of the previous step:

```
save_and_open_page
```

For example, in the feature spec shown earlier in this chapter, I could use Launchy to look at the results of my new user form:

spec/features/users_spec.rb

```
1  require 'spec_helper'
2
3  feature 'User management' do
4    scenario "adds a new user" do
5      admin = create(:admin)
6      sign_in admin
7
8      visit root_path
9      expect{
10       click_link 'Users'
11       click_link 'New User'
12       fill_in 'Email', with: 'newuser@example.com'
13       find('#password').fill_in 'Password', with: 'secret123'
14       find('#password_confirmation').fill_in 'Password confirmation',
15         with: 'secret123'
16       click_button 'Create User'
17     }.to change(User, :count).by(1)
18
19     save_and_open_page
20
21     # remainder of scenario
22   end
23 end
```

Remove the `save_and_open_page` line, of course, when you don't need it anymore. That one line has saved me untold hours of headache in my own specs.

A little refactoring

Before we move on, let's take another look at that feature spec for creating new users. There's at least one thing we can refactor. As you may recall, in chapter 7 we extracted the simulated user login into a helper macro. We can do the same thing for feature specs.

Why not just use the same technique we've used in controller specs? Because, at the feature level, we're testing that things work the way users would *interact* with them. This includes logging in! However, that doesn't mean we can't extract the login steps into a helper. Let's do that now:

spec/support/login_macros.rb

```
module LoginMacros
  # controller login helper omitted ...

  def sign_in(user)
    visit root_path
    click_link 'Log In'
    fill_in 'Email', with: user.email
    fill_in 'Password', with: user.password
    click_button 'Log In'
  end
end
```

And we can use the helper in our feature spec like this:

spec/features/users_spec.rb

```
feature 'User management' do
  scenario "adds a new user" do
    admin = create(:admin)
    sign_in admin

    # remaining steps omitted ...
  end
end
```

Including JavaScript interactions

So we've verified, with a passing spec, that our user interface for adding contacts is working as planned. Now let's test the About link in the application's navigation bar. While on the surface it seems incredibly basic, it in fact introduces a new wrinkle to our tests.

The spec looks something like this:

spec/features/about_us_spec.rb

```
require 'spec_helper'

feature "About BigCo modal" do
  scenario "toggles display of the modal about display" do
    visit root_path

    expect(page).to_not have_content 'About BigCo'
    expect(page).to_not \
      have_content 'BigCo produces the finest widgets in all the land'
```

```
10
11      click_link 'About Us'
12
13      expect(page).to have_content 'About BigCo'
14      expect(page).to \
15        have_content 'BigCo produces the finest widgets in all the land'
16
17      within '#about_us' do
18        click_button 'Close'
19      end
20
21      expect(page).to_not have_content 'About BigCo'
22      expect(page).to_not \
23        have_content 'BigCo produces the finest widgets in all the land'
24    end
25  end
```

Nothing too complex–but there's a problem. As-is, we're running the feature spec using Capybara's default web driver. This driver, Rack::Test, can't do JavaScript, so it ignores it. Therefore, the very first expectation in the example fails because, without JavaScript to hide the inline `#about_us` div in our *application.html.haml* file, Rack::Test sees the div and reports the failure.

Luckily, Capybara bundles support for the Selenium web driver out of the box. With Selenium, you can simulate more complex web interactions, including JavaScript, through your computer's installation of Firefox. Selenium makes this possible by running your test code through a lightweight web server, and automating the browser's interactions with that server.

Unfortunately, Selenium adds a non-Ruby dependency to the test suite: Firefox. The Mozilla Foundation's policy to frequently update the browser may, at some point, break your tests in some way. When that happens, try updating to the current version of the `selenium-webdriver` gem[42].

To use Selenium, we just need to make one small change to the example:

spec/features/about_us_spec.rb

```
1  require 'spec_helper'
2
3  feature "About BigCo modal" do
4    scenario "toggles display of the modal about display", js: true do
5      # the example ...
6    end
7  end
```

[42] http://rubygems.org/gems/selenium-webdriver

Notice what's different: We've added `js: true` to the scenario, to tell Capybara to use a JavaScript-capable driver (Selenium, by default). That's all there is to it! Run the spec again and watch as Firefox launches and runs through the steps of the scenario.

Admittedly, that's a very simple example. For the sake of demonstration, let's take a look at an example involving a little more user interaction, running through Selenium. Although the first scenario we created in this chapter doesn't *require* JavaScript to perform, let's enable JavaScript anyway and see what happens.

spec/features/users_spec.rb

```
feature 'User management' do
  scenario "adds a new user", js: true do
    # scenario steps ...
  end
end
```

We also need to configure Database Cleaner to help with database transactions in our tests. First, change RSpec's default settings for database transactions, and tell it to use DatabaseCleaner's `:truncation` method when running specs through the Selenium driver. Let's make the following changes to *spec/spec_helper.rb*:

spec/spec_helper.rb

```
RSpec.configure do |config|

  # earlier configurations omitted ...

  # Set config.use_transactional_fixtures to false
  config.use_transactional_fixtures = false

  config.before(:suite) do
    DatabaseCleaner.strategy = :truncation
  end

  config.before(:each) do
    DatabaseCleaner.start
  end

  config.after(:each) do
    DatabaseCleaner.clean
  end
```

Second, we need to monkey patch ActiveRecord to use threads. Add an additional file in *spec/support* with the following alterations to `ActiveRecord::Base`:

spec/support/shared_db_connection.rb

```
class ActiveRecord::Base
  mattr_accessor :shared_connection
  @@shared_connection = nil

  def self.connection
    @@shared_connection || retrieve_connection
  end
end
ActiveRecord::Base.shared_connection = ActiveRecord::Base.connection
```

Why is this necessary? The short answer is it's due to how Selenium handles database transactions. We need to share data state across the Selenium web server and the test code itself. Without DatabaseCleaner and the above patch, we're apt to get sporadic error messages resulting from tests not properly cleaning up after themselves.

For a more complete description of this setup, check out Avdi Grimm's Virtuous Code blog[43]. Thank you to reader Chris Peters for pointing this fix out to me.

With those changes, the feature spec will run through Firefox, and you're one step closer to a well-tested application.

Capybara drivers

So far, we've put two drivers to use in our feature specs. The default driver, RackTest, is a reliable solution for testing basic browser interactions. It's *headless*, so these interactions are all simulated in the background. Selenium is provided for more complicated interactions, including those requiring JavaScript or redirections (including redirections away from your application).

Selenium's added functionality comes at a price, however–you'll no doubt tire of waiting for Firefox to launch and run your specs every time, especially as your test suite grows. Fortunately there are headless options supporting JavaScript. Two popular headless drivers for Capybara include capybara-webkit[44] and Poltergeist[45]. Both of these drivers may require additional dependencies and can take some time to set up, but if your application has a lot of feature specs requiring more than the basics offered by RackTest it will be worth the extra setup time. Refer to Capybara's README[46] for details on setting up alternate drivers.

[43] http://devblog.avdi.org/2012/08/31/configuring-database_cleaner-with-rails-rspec-capybara-and-selenium/
[44] https://github.com/thoughtbot/capybara-webkit
[45] https://github.com/jonleighton/poltergeist
[46] https://github.com/jnicklas/capybara

Summary

This wasn't a long chapter–especially compared to the amount of time spent on controllers–but it does introduce a lot of new concepts while building upon what you've learned so far. In fact, its brevity can be attributed to the fact that it simply builds upon skills you acquired in the previous six chapters. Review it a few times if you have to, and keep practicing. If you get stuck, it's not against the rules to fire up your web browser and see if what you're expecting to happen in your tests is actually happening in the browser. (Launchy can help with this, too.)

At this point you've been exposed to the key tools and techniques you'll use to test your Rails applications. We've still got a few things to cover before we wrap up, though. In the next chapter we'll look at techniques to help keep your growing test suite running as quickly as possible.

Exercises

- Write some feature specs and make them pass! Start with simple user activities, moving on to the more complex as you get comfortable with the process.
- As you did with your controller specs, use this time to note places where your code could use refactoring. Again, if your app requires you to do a lot of setup to get everything just right for a test, it's a sign that you could be simplifying things in your code base. Clean up your code and run your feature specs again. Do they still pass?
- As you write the steps required for a given feature example, think about your users–they're the ones who work through these steps in their browsers when they need to get something done. Are there steps you could simplify–or even remove–to make the overall user experience for your application more pleasant?

9. Speeding up specs

Back in chapter 7, we did a round of refactoring on controller specs to make them easier to read and maintain. Specifically, we accomplished three tasks:

- We reduced redundancy with shared examples.
- We moved frequently-used functionality into helper macros.
- We built custom matchers to simplify expectations within our examples.

Now that we've got a relatively complete test suite, let's look again at how we can refactor–but this time for speed.

By *speed* I mean two things: One, of course, is the amount of time it takes your specs to run. Our little app's tests are already getting on the slow side. As it grows–assuming the test suite grows with it–that will change. The goal is to keep it manageable, without sacrificing the readability afforded us by RSpec. The second thing I mean by *speed* is how quickly you as a developer can create meaningful, clear specs. We'll touch on both of these aspects in this chapter. Specifically, we'll cover:

- RSpec's options for terse, but clean, syntax for shorter specs.
- Simplified specs with Shoulda's custom matchers.
- More efficient data for testing with mocks and stubs.
- Using tags to filter out slow specs
- Automating test execution and preloading Rails.
- Techniques for speeding up the suite as a whole.

Check out the *09_speedup* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 09_speedup origin/09_speedup
```

See chapter 1 for additional details.

Optional terse syntax

One critique of our specs so far might be that they're pretty wordy–we've been following some best practices and providing clear labels for each test, and one expectation per example. It's all been on purpose. However, RSpec provides techniques to continue these best practices while reducing your keystrokes. They may be used together to *really* streamline things, or individually to clean up longer-form specs.

let()

Up to this point we've been using `before :each` blocks to assign frequently-used test data to instance variables. An alternative to this, preferred by many RSpec users, is to use `let()`. `let()` gives us two advantages:

1. It *caches* the value without assigning it to an instance variable.
2. It is *lazily evaluated*, meaning that it doesn't get assigned until a spec calls upon it.

Here's how we can use `let()` in a controller spec:

spec/controllers/contacts_controller_spec.rb

```
1  require 'spec_helper'
2
3  describe ContactsController do
4    let(:contact) do
5      create(:contact, firstname: 'Lawrence', lastname: 'Smith')
6    end
7
8    # rest of spec file omitted ...
```

Then, instead of working with the contact via `@contact`, we can just use `contact` like so:

spec/controllers/contacts_controller_spec.rb

```
1  describe 'GET #show' do
2    it "assigns the requested contact to contact" do
3      get :show, id: contact
4      expect(:contact).to eq contact
5    end
6
7    it "renders the :show template" do
8      get :show, id: contact
9      expect(response).to render_template :show
10   end
11 end
```

However, this causes a problem in the example testing whether the controller's `destroy()` method actually deletes data from persistence. Here's the currently failing spec:

spec/controllers/contacts_controller_spec.rb

```
describe 'DELETE destroy' do
  it "deletes the contact" do
    expect{
      delete :destroy, id: contact
    }.to change(Contact,:count).by(-1)
  end
end
```

The count doesn't change, because the example doesn't know about `contact` until after we're in the `expect{}` Proc. To fix this, we'll just call `contact` before the Proc:

spec/controllers/contacts_controller_spec.rb

```
describe 'DELETE destroy' do
  it "deletes the contact" do
    contact
    expect{
      delete :destroy, id: contact
    }.to change(Contact,:count).by(-1)
  end
end
```

We could also use `let!()` (note the exclamation mark!), which forces `contact` to be assigned prior to each example. Or we could include `let()` within a `before` block–which may begin to defeat the purpose of using `let()` to begin with.

subject{}

`subject{}` lets you declare a test subject, then reuse it implicitly in any number of subsequent examples. Read on to see it in action.

it{} and specify{}

`it{}` and `specify{}` are synonymous–they are simple blocks that wrap an expectation. We've been using `it{}` since chapter 3, in a longer form. In other words, you could change

```
subject { build(:user, firstname: 'John', lastname: 'Doe') }

it 'returns a full name' do
  should be_named 'John Doe'
end
```

to

```
subject { build(:user, firstname: 'John', lastname: 'Doe') }
it { should be_named 'John Doe' }
```

And get the same results. Trivial here, perhaps, but as specs grow these one-liners can make a difference. Note, too, that even though we've been using the `expect` syntax in tests, these one-liners still use `should`. This is by design: As mentioned by RSpec's developers, `should` reads better in these examples.

Read your specs aloud as you write them, and use the term that makes the most sense–there are not hard rules about when to use one or the other.

Shoulda

Shoulda[47] is an extensive library of helpers to make testing common functionality a breeze. By including one additional gem, we can reduce some of our specs from three or four or five lines down to one or two.

`subject()`, `it{}` and `specify{}` really shine when used in conjunction with the `shoulda-matchers` gem. Include `shoulda-matchers` in the `:test` group of your Gemfile, and you'll automatically have access to a number of useful matchers–for example:

```
subject{ Contact.new }
specify { should validate_presence_of :firstname }
```

Nice and readable, with a good amount of coverage. We can also apply our own custom matchers to streamline even more. For example, the following custom matcher:

spec/models/contact_spec.rb

```
1 RSpec::Matchers.define :be_named do |expected|
2   match do |actual|
3     actual.name eq expected
4   end
5   description do
6     "return a full name as a string"
7   end
8 end
```

Can easily be called with the following `it{}` block:

```
it { should be_named 'John Doe' }
```

Yes, this example might be overkill, but hopefully it gives you an idea of the different ways you can streamline your specs–without sacrificing readability:

[47] http://rubygems.org/gems/shoulda

```
Contact
  should return a full name as a string
  should have 3 phones
  should require firstname to be set
  should require lastname to be set
```

And so on.

Mocks and stubs

Mocking and stubbing and the concepts behind them can be the subjects of lengthy chapters (if not whole books) of their own. Search them online and you'll inevitably come to an occasionally contentious debate on the right and wrong ways to use them. You'll also find any number of people attempting to define the two terms–to varying degrees of success. My best definitions of each:

- A **mock** is some object that represents a real object, for testing purposes. These are also known as *test doubles*. These are sort of what we've using Factory Girl to accomplish, with the exception that a mock doesn't touch the database–and thus takes less time to set up in a test.
- A **stub** overrides a method call on a given object and returns a predetermined value for it. In other words, a stub is a fake method which, when called upon, will return a real result for use in our tests. You'll commonly use this to override the default functionality for a method, particularly in database or network-intensive activity.

Here are a couple of loose examples:

- To create a mock contact, you can use the Factory Girl `build_stubbed()` method to generate a fully-stubbed fake, knowing how to respond to various methods like `firstname`, `lastname`, and `fullname`. It does not, however, persist in the database.
- To stub a method in the Contact model itself, you'd use a stub along the lines of `allow(Contact).to receive(:order).with('lastname,firstname').and_return([contact])`. In this case, we're overriding the `order` scope on the Contact model. We pass a string to specify the SQL order (in this case, the lastname and firstname fields), then tell it what we want back–a single-element array containing a `contact` we presumably created earlier in the spec.

Prior to RSpec 2.14, the stub from the previous paragraph would read like `Contact.stub(:order).with('lastname,firstname').and_return([contact])`. This older syntax works in 2.14; however, it's recommended you adopt the newer style moving forward.

In many cases, you're more likely to find RSpec's built-in mocking libraries[48] or an external library like Mocha[49] used in projects, or one of a number of other options available[50]. For the sake of a beginner's perspective here, they all operate similarly, albeit with tradeoffs.

It may make more sense to view these in the context of a controller spec.

spec/controllers/contacts_controller_spec.rb

```
1  describe 'GET #show' do
2    let(:contact) { build_stubbed(:contact,
3      firstname: 'Lawrence', lastname: 'Smith') }
4
5    before :each do
6      allow(Contact).to receive(:persisted?).and_return(true)
7      allow(Contact).to
8        receive(:order).with('lastname, firstname').and_return([contact])
9      allow(Contact).to
10       receive(:find).with(contact.id.to_s).and_return(contact)
11     allow(Contact).to receive(:save).and_return(true)
12
13     get :show, id: contact
14   end
15
16   it "assigns the requested contact to @contact" do
17     expect(assigns(:contact)).to eq contact
18   end
19
20   it "renders the :show template" do
21     expect(response).to render_template :show
22   end
23 end
```

Walking through the spec, we first use `let()` to assign a stubbed mock contact to `contact`. Then, we add some stubbed methods to both the Contact model and the `contact` instance. Since the controller will expect both Contact and `contact` to respond to several ActiveRecord methods, we need to stub the methods we'll be using in the actual controller, returning what we'd expect ActiveRecord to provide back to the controller. Finally, we use `it` blocks for the examples themselves, as we've been doing throughout the book. In this case, though, all of our test data are based on mocks and stubs, and not actual calls to the database or the Contact model itself.

On the plus side, this example is more isolated than specs we've written previously–its only concern is the controller method in question; it doesn't care about the model layer or the database or anything else. On the down side, this isolation is leading to additional code (and questionable readability) in the specs.

[48] http://rubydoc.info/gems/rspec-mocks/frames
[49] http://rubygems.org/gems/mocha
[50] https://www.ruby-toolbox.com/categories/mocking

With all that said, if you don't want to mess with mocks and stubs too much, don't worry–you can go a long way with using Ruby objects for basic stuff, and factories for more complex setups, as we have throughout this book. Stubs can also get you into trouble, as noted in the updated PeepCode series on RSpec[51]. One could easily stub out important functionality, resulting in tests that, well, don't actually test anything.

Unless things get very slow, or you need to test against data that is difficult to recreate (such as an external API or other web service, which we'll cover in a bit more practical tones in the next chapter) then prudent use of objects and factories may be all you need.

Automation with Guard and Spork

Forgetting to run specs early and often can result in lots of lost time. If you don't realize there's an issue somewhere, and keep piling new code on top of that issue, you may waste valuable minutes–or even hours. But switching to a terminal and running `rspec` from the command line can get tedious (and chip away at our time, too). Guard to the rescue!

Guard[52] watches files you specify, and does things based on what it sees. In our case, we want it to watch files in our *app* and *spec* directories, and run the relevant specs when those files change. For example, if I make a change to *app/models/contact.rb*, then *spec/models/contact_spec.rb* should run. If it fails, it should keep running until it passes.

To use Guard, first make sure `guard-rspec` is included in your Gemfile's `:test` and `:development` groups (see chapter 2). `guard-rspec` will include Guard itself.

Then create a `Guardfile` from the command line:

```
bundle exec guard init rspec
```

This will generate a `Guardfile` in your Rails application's root, serving as Guard's configuration for your app. It's pretty useful out of the box, but you'll probably want to tweak it to your own preferences. I typically set the following:

- `notification: false`: I prefer to keep an eye my specs running on a terminal window instead of receiving pop-ups messages.
- `all_on_start: false` and `all_on_pass: false`: I've been doing this for awhile; I know to run my full test suite before committing any changes I've made. If I want to run my specs at any time after firing up Guard I can just press *return*. Same with running all specs upon passing; I like having control of the situation.
- Run feature specs upon changes to views: Since I avoid RSpec view specs, I rely on Capybara feature specs to test this layer of my apps. Generally speaking, I don't run feature specs when changing my models or controllers. As with anything, though, it depends on the situation.

[51] https://peepcode.com/screencasts/rspec
[52] https://github.com/guard/guard

The generated `Guardfile` doesn't particularly lend itself to display in a book format. See the Guardfile in the sample source to see it more like it would look in an application.

Run `bundle exec guard` to get things going. Guard will run your full test suite, then dutifully observe for changes and run specs as needed. You can add other options as well–for example, I usually prefer to only run the full test suite on demand. The following additions to the Guardfile accomplish this:

Guardfile

```
guard 'rspec', version: 2, cli: '--color --format documentation',
  all_on_start: false, all_after_pass: false do
```

Guard's not just for watching and running your specs. It can compile Sass and LESS into CSS, run Cucumber features, minify JavaScript, run code metrics, reboot development servers, and more. Check out a full list of Guards[53] on GitHub. For more on Guard, check out the Railscasts episode[54] on the topic.

Once our tests get started, they run pretty quickly. However, at this point we've got a lag each time we start a test run–the lag caused by the Rails application needing to spin up each time. With Spork, we can limit the lag to just the first time we fire up the test suite–after that, specs will run with much more immediacy. Combined with Guard, Spork is one of the better ways to reduce your testing time.

There's already an excellent Railscasts episode on Spork[55] so I'm not going to cover it too in-depth here. However, you can see my Spork configuration in this chapter's sample source (refer to *Guardfile* and *spec/spec_helper.rb*). Be aware, since the coverage of Spork on Railscasts, Rails-specific features have been extracted to the spork-rails gem[56].

As of this writing, the version of Spork included in the `guard-spork` gem published on Rubygems isn't quite ready for Ruby 2.0.0. The GitHub-hosted version, however, works well. Include it in your Gemfile as follows:

Gemfile

```
group :development do
  gem 'spork-rails', github: 'sporkrb/spork-rails'
  # ...
end
```

You may also be interested in some alternative solutions to this problem like Zeus[57], Commands[58], and the promising Spring[59]. Like Spork, each has its own strengths and drawbacks–I've stuck with Spork here due to its longevity, relative ease of setup, and overall compatibility with various operating systems and Ruby versions.

[53] https://github.com/guard
[54] http://railscasts.com/episodes/264-guard
[55] http://railscasts.com/episodes/285-spork
[56] https://github.com/sporkrb/spork-rails
[57] https://github.com/burke/zeus
[58] https://github.com/rails/commands
[59] https://github.com/jonleighton/spring

Tags

Whether or not you opt to add Guard to your workflow, RSpec's tags feature[60] can help you fine-tune which specs to run at a given time. To apply a tag, add it to a given example:

```
it "processes a credit card", focus: true do
  # details of example
end
```

You can then run only the specs with the `focus` tag from the command line:

```
$ bundle exec rspec . --tag focus
```

You can also configure RSpec to only run (or never run) examples with specific tags; for example:

spec/spec_helper.rb

```
RSpec.configure do |c|
  c.filter_run focus: true
  c.filter_run_excluding slow: true
end
```

This is particularly useful when using Guard, as you can turn a given tag on or off in your `spec_helper.rb` file, allow Guard to reload itself, and keep working. I don't use this feature often, but find it invaluable when I need it.

Other speedy solutions

Remove unnecessary tests

If a test has served its purpose, and you're confident you don't need it for regression testing, delete it. If you *do* want to hold onto it for some reason, mark it as a pending spec:

```
1 it "loads a lot of data" do
2   pending "no longer necessary"
3   # your spec's code; it will not be run
4 end
```

I recommend this over commenting out the test–since pending specs are still listed when you run the test suite you'll be less apt to forget they're there. That said, I ultimately recommend just deleting the unnecessary code–but only when you're comfortable doing so.

[60] https://www.relishapp.com/rspec/rspec-core/v/2-4/docs/command-line/tag-option

Take Rails out of the equation

The changes we've made above will all play a part in reducing the amount of time it takes the suite to run, but ultimately one of the biggest slowdowns is Rails itself–whenever you run tests, some or all of the framework needs to be fired up. If you *really* want to speed up your test suite, you can go all out and remove Rails from the equation entirely. Whereas Spork still loads the framework–but limits itself to loading once–these solutions go one step further.

This is a little more advanced than the scope of this book, as it requires a hard look at your application's overall architecture. It also breaks a personal rule I have when working with newer Rails developers–that is, avoid breaking convention whenever possible. If you want to learn more, though, I recommend checking out Corey Haines' talk on the subject[61] and the Destroy All Software[62] screencast series from Gary Bernhardt.

Summary

We looked at some pretty weighty topics in this chapter. Up until now, I didn't talk about varying techniques to get the testing job done–but now you've got options. You can choose the best way for you and your team to provide clear documentation through your specs–either by using a verbose technique as we did in chapter three, or in a more terse fashion as shared here. You can also choose different ways to load and work with test data–mocks and stubs, or factories, or basic Ruby objects, or any combination thereof. Finally, you now know a few different techniques for loading and running your test suite. You're on your way to making RSpec your own.

We're in the home stretch now! Just a few more things to cover, then we'll wrap up with some big picture thinking on the testing process in general and how to avoid pitfalls. First, let's look at some of the corners of a typical web application we *haven't* tested yet.

Exercises

- Find specs in your suite that could be cleaned up with `let()`, `subject{}`, and `it{}`. By how much does this exercise reduce your spec's footprint? Is it still readable? Which method–terse or verbose–do you prefer? (Hint: There's really no right answer for that last question.)
- Install `shoulda-matchers` in your application and find places you can use it to clean up your specs (or test things you haven't been testing). Look into the gem's source on GitHub to learn about all of the matchers Shoulda offers to RSpec users.
- Using RSpec tags, identify your slow specs. Run your test suite including and excluding the tests. What kind of performance gains do you see?

[61] http://confreaks.com/videos/641-gogaruco2011-fast-rails-tests
[62] https://www.destroyallsoftware.com/screencasts

10. Testing the rest

At this point we've got decent coverage across the address book application. We've tested our models and controllers, and also tested them in tandem with views via feature specs. For this basic application we should be covered pretty well with these core testing techniques. However, many Rails applications (including yours, probably) aren't this simple. Maybe your app sends email to users, or interacts with an external web service, or handles file uploads. Maybe it performs certain functionality based on the date or time. We can test these facets, too!

In this chapter we'll survey:

- How to test for email delivery.
- How to test file uploads.
- Manipulating the time within specs.
- Testing against external web services.
- Testing rake tasks.

No code samples for this chapter–everything I tried to come up with as a feature for the address book seemed contrived, though I'm not ruling out some code samples for a future release of the book.

Testing email delivery

Testing that your application's mailers are doing their job is relatively easy–all it takes is another gem and a few more lines of configuration.

The gem is Email Spec[63]–a useful set of custom matchers to test a given message's recipients, subject, headers, and content. Once you've added the gem to your Gemfile's `:test` group and run `bundle install`, you'll just need to add a few more configuration lines to *spec/spec_helper.rb*:

spec/spec_helper.rb

```
1 require "email_spec"
2 config.include(EmailSpec::Helpers)
3 config.include(EmailSpec::Matchers)
```

With these lines you may now add expectations like the following:

[63] http://rubygems.org/gems/email_spec

```
# some setup done to trigger email delivery ...

expect(open_last_email).to be_delivered_from sender.email
expect(open_last_email).to have_reply_to sender.email
expect(open_last_email).to be_delivered_to recipient.email
expect(open_last_email).to have_subject message.subject
expect(open_last_email).to have_body_text message.message
```

where `open_last_email` is a helper that opens the most recently-sent email and gives you access to its attributes. As outlined in the library's documentation[64], you can also create a new mail object and work directly with it:

```
email = MessageMailer.create_friend_request("aaron@everydayrails.com")
expect(email).to deliver_to("aaron@everydayrails.com")
expect(email).to have_subject "Friend Request"
```

As you can see, the custom matchers provided by Email Spec are nice and readable–see a complete list of of matchers in the documentation. While there, take a look at the list of helpers made available to you. I like to use `open_last_email` in particular when testing mail delivery at the integration (feature) level. In general, Email Spec works great at the model and controller levels as well–use it where it makes the most sense within your application.

For more on Email Spec and testing mail delivery in general, I defer once again to the Railscasts episode, "How I Test"[65]. It's required viewing, as far as I'm concerned, for anyone wanting to get better with testing mail in RSpec.

Testing file uploads

Making sure file uploads worked the way I'd intended was a sticking point in my testing routine for a long time. In particular, how does a fake file get included into a spec? Where does it get stored in the meantime? Even though Rails provides a means of uploading files from your *fixtures* directory, I've found it to be hit or miss and tend to use this straightforward method. Place a small, dummy file (ideally representative of your real-world data) in your *spec/factories* directory. Then you can refer to it in a factory like so:

[64] http://rubydoc.info/gems/email_spec/1.5.0/frames
[65] http://railscasts.com/episodes/275-how-i-test

```
FactoryGirl.define do
  factory :user do
    sequence(:username) { |n| "user#{n}"}
    password 'secret'
    avatar { File.new("#{Rails.root}/spec/factories/avatar.png") }
  end
end
```

If your model *requires* the attached file to be present, you'll probably want to make sure it gets stored when using a factory to generate test data.

More importantly, you can also access the file explicitly in specs, such as the following feature example:

```
it "creates a new user" do
  visit new_user_url
  fill_in 'Username', with: 'aaron'
  fill_in 'Password', with: 'secret'
  attach_file 'Avatar',
    File.new("#{Rails.root}/spec/factories/avatar.png")
  click_button 'Sign up'
  expect(User.last.avatar_file_name).to eq 'avatar.png'
end
```

Using the factory above, we can also test this at the controller level like this:

```
it "uploads an avatar" do
  post :create, user: create(:user)
  expect(assigns(:user).avatar_file_name).to eq 'avatar.png'
end
```

Assuming you're using a popular file upload library like Paperclip or Carrierwave, though, you're apt to have some built-in testing conveniences at your disposal–consult the documentation for the gem in question.

Testing the time

What if your application has expectations based on the time or date? For example, say we want to wish visitors a Happy New Year when they visit our site, but only on January 1. We can use Timecop[66] to freeze time, making it possible to test such things without resorting to heavy-duty Ruby date manipulation. All you need to do is include the Timecop gem in your Gemfile, then use it like I am in this hypothetical feature spec:

[66] http://rubygems.org/gems/timecop

```
it "wishes the visitor a Happy New Year on January 1" do
  Timecop.travel Time.parse("January 1")
  visit root_url
  expect(page).to have_content "Happy New Year!"
  Timecop.return
end
```

Take note of the call to `Timecop.return` in these code samples. This resets the clock to your system's time and helps RSpec properly report the amount of time your tests take to run.

Timecop is also useful in situations where you need to impose a deadline–for example, maybe you need to make sure people have filed their taxes on time:

```
it "doesn't allow taxpayers to file after April 15" do
  Timecop.travel Time.parse("April 16")
  visit tax_submit_form
  expect(page).to have_content "Sorry, you're too late!"
  Timecop.return
end
```

or

```
it "gives taxpayers up until the 15th to file" do
  Timecop.travel Time.parse("April 15")
  visit tax_submit_form
  expect(page).to have_content "There's still time to file, but hurry!"
  Timecop.return
end
```

That's how I usually use Timecop, but another common usage is when you want to *stop* time during the test. For example, maybe you want to be really sure Rails' default timestamps are working. You could do something like this in a model spec:

```
it "stamps the model's created at with the current time" do
  Timecop.freeze
  user = create(:user)
  expect(user.created_at).to eq Time.now
  Timecop.return
end
```

Without `Timecop.freeze` in the example, the split-second difference between when the data was persisted and when the spec checks its value would be just enough to cause it to fail.

Testing web services

Full disclosure: I have to test an application's interaction with web services very infrequently. Most of the work I do in my day job is fairly isolated from the rest of the world. However, I have worked on a handful of projects requiring things like credit card processing, and can tell you that while it's technically possible to test these at the integration level, using techniques I've already shared, it's slow and prone to errors. It's far better to stub out these services. Luckily, there are a few utilities that make this very easy.

Allow me to, once again, direct you to a couple more Railscasts: First is a free episode on using Fakeweb[67] to–you guessed it–fake web requests without having to write complex stubs. The second episode requires a Railscast Pro subscription, but is essential if you need to test against web services. VCR[68] works atop an HTTP library (such as Fakeweb) and makes recording and reusing web service interactions incredibly simple and fast. (The VCR documentation[69] is also pretty solid, if you'd prefer a free resource.)

Testing rake tasks

If you've been developing in Rails for awhile you've probably written at least one Rake command line utility for your application. I often use Rake tasks for things like legacy data transfers or scheduled operations. Legacy transfers in particular can get pretty gnarly, so just like the rest of my code I like to build tests to make sure I won't get any surprises.

In my experience, the best way to do this is to abstract any code you've got in a given Rake task into a class or module, then call that method within the task. For example, imagine that we've got a Rake task to move information from a legacy `Person` class to the `Contact` class we've been using throughout the book. A procedural approach to this might look something like this:

```
1  namespace :legacy do
2    desc "Move Persons to Contacts"
3    task person: :environment do
4      Person.all.each do |person|
5        Contact.create!(
6          firstname:  person.firstname,
7          lastname:   person.lastname,
8          email:      person.email
9        )
10     end
11   end
12 end
```

In this case, I might create a `Legacy` class *lib/legacy.rb* and move the bulk of the task to a class method within it:

[67] http://railscasts.com/episodes/276-testing-time-web-requests
[68] http://railscasts.com/episodes/291-testing-with-vcr
[69] https://www.relishapp.com/myronmarston/vcr/docs

```
class Legacy
  def self.move_people
    Person.all.each do |person|
      Contact.create!(
        firstname:  person.firstname,
        lastname:   person.lastname,
        email:      person.email
      )
    end
  end
end
```

And update my original Rake task:

```
namespace :legacy do
  desc "Move Persons to Contacts"
  task person: :environment do
    Legacy.move_people
  end
end
```

Now I can easily test the task by testing the `Legacy` class. To mirror the application structure, let's first create the directory *spec/lib*, then add *legacy_spec.rb* to it and test:

```
require 'spec_helper'

describe Legacy do
  it 'creates a contact from a person'
  # etc.
end
```

You can use the same techniques we've covered throughout the book to test Rake-related code just like you would any other code in your application.

Summary

Even though things like email, file uploads, timestamps, web services, and utility tasks may be on the fringes of your application, take the time to test them as needed–because you never know, one day that web service may become more central to your app's functionality, or your next app may rely heavily on email. There's never a bad time to practice, practice, practice.

You now know how to test everything *I* test on a regular basis. It may not always be the most elegant means of testing, but ultimately it provides me enough coverage that I feel comfortable adding features to my projects without the fear of breaking functionality–and if I *do* break something, I can use my tests to isolate the situation and fix the problem accordingly.

As we wind down our discussion of RSpec and Rails, I'd like to talk about how to take what you know and use it to develop software in a more *test-driven* fashion. That's what we'll cover in the next chapter.

Exercises

- If your application has any mailer functionality, get some practice testing it now. Common candidates might be password reset messages and notifications.
- Does your application have any file upload functionality or time-sensitive functions? Again, it's a great idea to practice testing these functions, using the utilities shared in this chapter. It's easy to forget about these requirements until one early morning or late night when they don't work.
- Have you written any specs against an external authorization service, payment processor, or other web service? How could you speed things up with VCR and Fakeweb?

11. Toward test-driven development

Whew. We've come a long way with our address book application. At the beginning of the book it had the functionality we were after, but zero tests. Now it's reasonably tested, and we've got the skills necessary to go in and plug any remaining holes.

But have we been doing test-driven development?

Strictly speaking, no. The code existed long before we added a single spec. What we've been doing is closer to *exploratory* testing–using tests to better understand the application. To legitimately practice TDD, we'll need to change our thinking–tests come first, then the code to make those tests pass, then refactoring to make our code more resilient for the long haul. Let's try it now in our sample application!

Check out the *11_tdd* branch of the sample source to see the completed code for this chapter. Using the command line, type

```
git checkout -b 11_tdd origin/11_tdd
```

See chapter 1 for additional details.

Defining a feature

The fictitious company for which we've been developing a contacts directory throughout the book now needs a place to post news releases, and has requested a link for these items be added to the application's menu bar. In the interest of keeping this exercise simple, let's say that anybody with an account can add a news release, and they're available immediately for site guests to view.

Those are the two basic scenarios:

- As a user, I want to add a news release so that the world can see how great our company is.
- As a site visitor (guest), I want to read news releases so that I can learn more about the company.

In this chapter, we'll tackle that first scenario; I'll let you take on the second as an exercise on your own. There are, of course, several other scenarios to consider here–updating or deleting news releases, for example, or establishing an editorial workflow in which administrators must approve news prior to making it public–but I'm going to leave those to you, too, should you want some additional practice.

With the initial story in mind, let's begin. Fire up Guard, and if necessary press `<return>` to run all specs and make sure nothing's broken before we start adding features. If anything isn't passing, use the skills you've learned throughout this book to make it pass–it's important to work from a clean slate before starting further development on a project.

Next, we'll outline our work in a new feature spec. In your editor, add a new file:

spec/features/news_releases_spec.rb

```
require 'spec_helper'

feature "News releases" do
  context "as a user" do
    scenario "adds a news release"
  end

  context "as a guest" do
    scenario "reads a news release"
  end
end
```

Save the file, and Guard will run just its specs automatically. As you should hopefully expect by now, we get the following feedback from RSpec:

```
2 examples, 0 failures, 2 pending
```

Let's add some steps to that first scenario:

spec/features/news_releases_spec.rb

```
require 'spec_helper'

feature "News releases" do
  context "as a user" do
    scenario "adds a news release" do
      user = create(:user)
      sign_in(user)
      visit root_path
      click_link "News"

      expect(page).to_not have_content "BigCo switches to Rails"
      click_link "Add News Release"

      fill_in "Date", with: "2013-07-29"
      fill_in "Title", with: "BigCo switches to Rails"
      fill_in "Body",
        with: "BigCo has released a new website built with open source."
      click_button "Create News release"

      expect(current_path).to eq news_releases_path
      expect(page).to have_content "Successfully created news release."
      expect(page).to have_content "2013-07-29: BigCo switches to Rails"
    end
```

```
24    end
25
26    # ...
27  end
```

From red to green

Check back in your terminal–Guard should have noticed the change in the spec and run it automatically. And we've got a failure! Remember, in test-driven development, this is a good thing–it gives us a goal to work toward. RSpec has given us a clear indication of what failed:

```
1) News releases as a user adds a news release
   Failure/Error: click_link "News"
   Capybara::ElementNotFound:
     Unable to find link "News"
```

Let's fix it and move forward. Open our application's layout view template and add the missing link to the navbar:

app/views/layouts/application.html.erb

```
<%= menu_item "News", news_releases_path %>
```

`menu_item` is a helper method provided by the `twitter_bootstrap_rails` gem.

We've made a couple of design decisions here–the link guests will click on will be labeled *News*, and we'll need a *news_releases_path* to render a list of available news releases. In most cases, these decisions will have been made up front–it will be up to you to convert them into an automated test with Capybara.

Return to the terminal, and–wait, why didn't the feature spec automatically run with the file change? Because Guard isn't set to watch to the layout file we just edited, it doesn't trigger any specs to run. We've got a few options to consider:

- We could edit the `Guardfile` to watch the template and run one or more specs.
- We could just press return to force Guard to run *all* specs, including slow ones.
- We could take advantage of RSpec's tagging feature to only run specs with the *focus* tag set to *true*.

I like the last option best. In fact, I'm going to take advantage of it throughout the rest of the chapter, as tests dart up and down other levels of the application. Referring back to chapter 9 if necessary, add a `filter_run` configuration line to *spec/spec_helper.rb*:

spec/spec_helper.rb

```
Spork.prefork do
  # ...
  RSpec.configure do |config|
    # ...
    config.filter_run focus: true
  end
end
```

and a `focus` tag to the spec we're working on:

spec/features/news_releases_spec.rb

```
feature "News releases", focus: true do
  # ...
```

Give Guard a moment to reload itself with the new configuration in mind, and then watch only the scenario we're working on run. In this case saving the feature file when we added the `focus` tag triggered the spec to run; however, if we press return at the `guard(main)>` prompt we can run *just* the spec(s) we're interested in and not those for other components of the site. We'll come back to those later before calling the feature complete.

If for whatever reason it doesn't seem like Guard is automatically reloading your RSpec settings, just enter `reload` at the prompt.

Moving on, back in the terminal we've still got red–but it's a new failure:

```
1) News releases as a user adds a news release
   Failure/Error: sign_in(user)
   ActionView::Template::Error:
     undefined local variable or method `news_releases_path' for
     #<#<Class:0x007fb90d506b08>:0x007fb90d50e498>
```

Rails is complaining, by way of RSpec, that we haven't defined a route for `news_releases_path`. Decision time again–do we:

- Explicitly add the route to the application, thus doing the simplest thing possible to make the spec pass?
- Take advantage of Rails' scaffolding to generate the route for us, as well as a bunch of other code we may or may not need?

In this case, I'm going with the second option. I know that for this feature to eventually be complete we're going to need to allow not only listing existing news releases, but also showing, adding, editing, and deleting them. A scaffold provides solid code to work from for all of actions, allowing us to spend less time writing boilerplate code and more time fine-tuning for our own application.

Open a new terminal into the application, if necessary, and generate the scaffold now:

```
$ rails g scaffold news_release title released_on:date body:text
```

Among all the files generated–some of which we'll use, some of which we won't–observe the following:

```
invoke    rspec
create      spec/models/news_release_spec.rb
invoke      factory_girl
create        spec/factories/news_releases.rb
...
invoke    rspec
create      spec/controllers/news_releases_controller_spec.rb
```

Way back in chapter 2, we configured the scaffold generator to provide these starter files for model and controller specs, and an initial factory with which to work as we plow forward.

Back to Guard, it looks like editing the route (by way of generating a scaffold) ran our focused spec again, and now complains that it can't find a table called `news_release`. No problem–we just need to run migrations:

```
$ rake db:migrate db:test:clone
```

Another new failure–but believe it or not, we're making progress:

```
Failures:

  1) News releases as a user adds a news release
     Failure/Error: click_link "Add News Release"
     Capybara::ElementNotFound:
       Unable to find link "Add News Release"
     # ./spec/features/news_releases_spec.rb:10:in \
       `block (3 levels) in <top (required)>'
```

Looking through the scenario, the failure occurs *after* clicking the *News* link in the navbar, and *after* RSpec confirms that the returned page–that is, the page rendered by *news_releases_path*–does not contain the title of the news release we haven't added yet. This new failure must be due to something in the *index* template for news releases. Load that file, and sure enough, the link to creating news releases, as generated by the scaffold, has different wording than what we've got in the spec. Let's fix that now, and add a little Bootstrap style while we're there:

app/views/news_releases/index.html.erb

```
<%= link_to 'Add News Release', new_news_release_path,
  class: 'btn btn-primary' %>
```

The next error suggests that the *Date* text field can't be found. Again, easily fixed:

app/views/news_releases/_form.html.erb

```
<%= f.label :released_on, 'Date' %><br>
<%= f.text_field :released_on %>
```

Looks like the spec is passing through the form now–the next error suggests we need to set the desired path to redirect to upon successfully posting a news release:

```
1) News releases as a user adds a news release
   Failure/Error: expect(current_path).to eq news_releases_path

     expected: "/news_releases"
          got: "/news_releases/1"

     (compared using ==)
   # ./spec/features/news_releases_spec.rb:18:in `block (3 levels)
     in <top (required)>'
```

This is because our preferred behavior deviates from the Rails scaffold's defaults. It's easily addressed by fixing the controller:

app/controllers/news_releases_controller.rb

```
def create
  @news_release = NewsRelease.new(news_release_params)

  respond_to do |format|
    if @news_release.save
      format.html { redirect_to news_releases_url,
          notice: 'News release was successfully created.' }
  # ...
```

We're getting close. To address the next failing step, change the value for the `:notice` symbol to *Successfully created news release.*

app/controllers/news_releases_controller.rb

```
def create
  @news_release = NewsRelease.new(news_release_params)

  respond_to do |format|
    if @news_release.save
      format.html { redirect_to news_releases_url,
          notice: 'Successfully created news release.' }
  # ...
```

One last step to complete the scenario and make it pass!

```
Failures:

  1) News releases as a user adds a news release
     Failure/Error: expect(page).to have_content \
       "2013-07-29: BigCo switches to Rails"
     # ...
     # ./spec/features/news_releases_spec.rb:20:in
       `block (3 levels) in <top (required)>'
```

The simplest way to make this pass is to add some code to the view template. Soon we'll consider ways to refactor this into cleaner code, but in the meantime:

app/views/news_releases/index.html.erb

```
1  <h1>News releases</h1>
2
3  <ul>
4    <% @news_releases.each do |news_release| %>
5      <li>
6        <%= link_to "#{news_release.released_on.strftime('%Y-%m-%d')}:
7          #{news_release.title}", news_release %>
8      </li>
9    <% end %>
10 </ul>
11
12 <p>
13   <%= link_to 'Add News Release', new_news_release_path,
14     class: 'btn btn-primary' %>
15 </p>
```

And with that, the first scenario passes!

Nice work, but we're not done–we've got some things we can clean up. We've arrived at the *refactor* stage of *red-green-refactor*. It can get quite involved if you allow it, but here we'll keep things relatively simple, as heavy-duty refactoring is beyond the scope of this book.

We can tackle at least one item: As alluded to earlier, the line we added to link to an individual news release is pretty ugly. A very simple refactoring might move this to the NewsRelease model. Of course, we should write a quick test for that first, in the NewsRelease model:

spec/models/news_release_spec.rb

```
require 'spec_helper'

describe NewsRelease, focus: true do
  it "returns the formatted date and title as a string" do
    news_release = NewsRelease.new(
      released_on: '2013-07-31',
      title: 'BigCo hires new CEO')
    expect(news_release.title_with_date).to \
      eq '2013-07-31: BigCo hires new CEO'
  end
end
```

And make it pass by updating the model:

app/models/news_release.rb

```
class NewsRelease < ActiveRecord::Base
  def title_with_date
    "#{released_on.strftime('%Y-%m-%d')}: #{title}"
  end
end
```

And a quick edit to our view, to use the new method:

app/views/news_releases/index.html.erb

```
<%= link_to news_release.title_with_date, news_release %>
```

And we're still green. That's they key to the refactoring step: *Any changes you make should result in the tests still passing*. You'll also find yourself darting up and down, from feature spec down to model, controller, and/or library. It depends on where it makes the most sense to keep the code in your application.

As long as we've got the model spec for NewsRelease open, let's think a bit about what other requirements the model might have. A couple of validations come to mind–in particular, a news release isn't of much use without a release date, title, or body. Let's throw in some expectations, using the matchers provided by the custom matchers provided by Shoulda (see chapter 9):

spec/models/news_release_spec.rb

```
it { should validate_presence_of :released_on }
it { should validate_presence_of :title }
it { should validate_presence_of :body }
```

Add the validations to the NewsRelease model to make these pass and continue.

There's one more aspect of this new feature we haven't paid attention to yet: Authentication. We know that a user can log in and add a news release with no problem, but what about guests? It should be no surprise that, if we were to comment out the two lines of our scenario that handles creating a test user and logging in with that user, it would still pass. That's not good.

Of course, we need to apply our authentication filter to the news releases controller. Instead of meticulously testing every controller method, let's focus on the important ones here: We want to make sure guests are denied access to the `new` and `create` methods, so let's turn our attention to the scaffolded controller spec we generated a little while ago. The scaffold generates a lot of extra code, but we can delete it all and replace it with the following:

spec/controllers/news_releases_controller_spec.rb

```
require 'spec_helper'

describe NewsReleasesController, focus: true do
  describe 'GET #new' do
    it "requires login" do
      get :new
      expect(response).to require_login
    end
  end

  describe "POST #create" do
    it "requires login" do
      post :create, news_release: attributes_for(:news_release)
      expect(response).to require_login
    end
  end
end
```

`require_login` is a custom matcher we created back in chapter 7.

I'm going to add a factory to help that second expectation pass:

spec/factories/news_releases.rb

```
require 'faker'

FactoryGirl.define do
  factory :news_release do
    title "Test news release"
    released_on 1.day.ago
    body { Faker::Lorem.paragraph }
  end
end
```

The spec dutifully fails:

```
Failures:

  1) NewsReleasesController GET #new requires login
     Failure/Error: expect(response).to require_login
       expected to require login to access the method
     # ./spec/controllers/news_releases_controller_spec.rb:7:in
       `block (3 levels) in <top (required)>'

  2) NewsReleasesController POST #create requires login
     Failure/Error: expect(response).to require_login
       expected to require login to access the method
     # ./spec/controllers/news_releases_controller_spec.rb:14:in
       `block (3 levels) in <top (required)>'
```

We'll add the authentication filter to lock things down, and pass the specs:

app/controllers/news_releases_controller.rb

```
class NewsReleasesController < ApplicationController
  before_action :authenticate, except: [:index, :show]
  # ...
```

Cleaning up

The tests are passing, and the new feature is implemented. Before we wrap things up, though, we've got a little more code cleanup to do. In this case, our initial decision to go with a scaffold has added a lot of extra code our application doesn't need right now, so we might as well delete it.

Among the files we *have* added or modified, the following sit untouched since we generated the scaffold:

- *app/assets/javascripts/news_releases.js.coffee*
- *app/assets/stylesheets/news_releases.css.scss*
- *app/assets/stylesheets/scaffolds.css.scss*
- *app/helpers/news_releases_helper.rb*

They're not used, so let's delete them. We can always add them back later if necessary.

Finally, we need to make sure our new feature hasn't interfered with previous work. Remove the `focus` tag from our new specs in *spec/features/news_releases_spec.rb*, *spec/models/news_release_spec.rb*, and *spec/controllers/news_releases_controller_spec.rb*. Get rid of the tag filter in *spec/spec_helper*:

spec/spec_helper.rb

```
# get rid of this line or at least comment it out:
# config.filter_run focus: true
```

Check back in the terminal window in which Guard is running, and press `<return>` to run the whole test suite. Looks good! All of our specs are passing, except for that pending spec to drive guests reading news releases–and you're handling that one, right?

One more thing: Even though everything looks good in the test suite, be sure to spot-check your work in your browser before you commit (and especially before you deploy). You're apt to notice things you wouldn't have otherwise. In fact, we've got a rather glaring issue to address in that next scenario: The *Add News Release* button on our list of news releases is visible to guests and signed-in users alike! Maybe, when you get to work on that next scenario, you might want to `expect(page).to_not have_content 'Add News Release'` (hint).

Just kidding–if you check out the source for this chapter on GitHub, you'll find that I've begun a scenario for you to make pass.

Summary

That's how I use RSpec to drive new features in my own Rails applications. While it may seem like a lot of steps on paper, in reality it's not that much extra work in the short term–and in the long term, writing tests and refactoring early saves much more time in the long haul. You've got the tools you need to do the same in your own projects!

Exercises

- If you're following along with the sample code, go ahead and work through that next scenario. Make the steps I've written in the pending scenario pass! Remember to consider specs at other levels of the application as needed–that is, are there any additional cases which might be more straightforward to test at the controller or model level?
- For further practice, see what you can do with other aspects of this general feature–what about editing and deleting news releases?

12. Parting advice

You've done it! If you've been adding tests to your application as you worked through the patterns and techniques outlined in this book, you should have the beginnings of a well-tested Rails application. I'm glad you've stuck with it this far, and hope that by now you're not only comfortable with tests, but maybe even beginning to think like a true test-driven developer and using your specs to influence your applications' actual under-the-hood design. And dare I say, you might even find this process fun!

To wrap things up, here are a few things to keep in mind as you continue down this path:

Practice testing the small things

Diving into TDD through complex new features is probably not the best way to get comfortable with the process. Focus instead on some of the lower-hanging fruit in your application. Bug fixes, basic instance methods, controller-level specs–these are typically straightforward tests, usually requiring a little bit of setup and a single expectation. Just remember to write the spec before tackling the code!

Be aware of what you're doing

As you're working, think about the processes you're using. Take notes. Have you written a spec for what you're about to do? Does the spec cover edge cases and fail states? Keep a mental checklist handy as you work, making sure you're covering what needs to be covered as you go.

Short spikes are OK

Test-driven development doesn't mean you can only write code once it's got a test to back it. It means you should only write *production* code after you've got the specs. Spikes are perfectly fine! Depending on the scope of a feature, I'll often spin up a new Rails application to tinker with an idea. I'll typically do this when I'm experimenting with a library or some relatively wholesale change. For example, I once worked on a data mining application in which I needed to completely overhaul the application's model layer, without adversely affecting the end user interface. I knew what my basic model structure would look like, but I needed to tinker with some of the finer points to fully understand the problem. By spiking this in a standalone application, I was free to hack and experiment within the scope of the actual problem I'm trying to solve–then, once I'd determined that I understood the problem and have a good solution for it, I opened up my production application, wrote specs based on what I learned in my tests, then wrote code to make those specs pass.

For smaller-scale problems I'll work in a feature branch of the application, doing the same type of experimentation and keeping an eye on which files are getting changed. Going back to my data mining project, I also had a feature to add involving how users would view already-harvested data. Since I already had the data in my system, I created a branch in Git and spiked a simple solution to make sure I understood the problem. Once I had my solution, I commented out what I'd written, wrote my specs, and then systematically reapplied my work.

As a general rule, I try to retype my work as opposed to just uncommenting it (or copying and pasting); I often find ways to refactor or otherwise improve what I did the first time.

Write a little, test a little is also OK

If you're still struggling with writing specs first, it is acceptable to code, then test; code, then test–as long as the two practices are closely coupled. I'd argue, though, that this approach requires more discipline than just writing tests first (after untested spikes). In other words, while I say it's OK, I don't think it's *ideal*. But if it helps you get used to testing then go for it.

Strive to write feature specs first

Once you get comfortable with the basic process and the different levels at which to test your application, it's time to turn everything upside down: Instead of building model specs and then working up to controller and feature specs, you'll *start* with feature specs, thinking through the steps an end user will follow to accomplish a given task in your application. This is essentially what's referred to as *outside-in* testing, and is the general approach we followed in chapter 11.

As you work to make the feature spec pass, you'll recognize facets that are better-tested at other levels–in the previous chapter, for example, we tested validations at the model level; authorization nuances at the controller level. A good feature spec can serve as an outline for all of the tests pertaining to a given feature, so learning to begin by writing them is a valuable skill to have.

Make time for testing

Yes, tests are extra code for you to maintain; that extra care takes time. Plan accordingly, as a feature you may have been able to complete in an hour or two before might take a whole day now. This especially applies when you're getting started with testing. However, in the long run you'll recover that time by working from a more trustworthy code base.

Keep it simple

If you don't get some aspects of testing right away–in particular, feature specs or mocking and stubbing–don't worry about it. They require some additional setup and thinking to not just work, but actually test what you need to test. Don't stop testing the simpler parts of your app, though–building skills at that level will help you grasp more complicated specs sooner rather than later.

Don’t revert to old habits!

It’s easy to get stuck on a failing test that shouldn’t be failing. If you can’t get a test to pass, make a note to come back to it–*and then come back to it*. Remember, point-and-click testing in your browser will only get slower and more tedious as your application grows. Why not use the time you’ll save on getting better at writing specs?

Use your tests to make your code better

Don’t neglect the *Refactor* stage of *Red-Green-Refactor*. Learn to listen to your tests–they’ll let you know when something isn’t quite right in your code, and help you clean house without breaking important functionality.

Sell others on the benefits of automated testing

I still know far too many developers who don’t think they have time to write test suites for their applications. (I even know a few who think that being the only person in the world who understands how a brittle, spaghetti-coded application works is actually a form of job security–but I know you’re smarter than that.) Or maybe your boss doesn’t understand why it’s going to take a little longer to get that next feature out the door. Take a little time to educate these people. Tell them that tests aren’t just for development; they’re for your applications’ long-term stability and everyone’s long-term sanity. Show them how the tests work–I’ve found that showing off a feature spec with JavaScript dependencies, as we put together in chapter 8, provides a wow factor to help these people understand how the extra time involved in writing these specs is time well-spent.

Keep practicing

Finally, it might go without saying, but you’ll get better at the process with lots of practice. Again, I find throwaway Rails applications to be great for this purpose–create a new app (say, a blogging app or to-do list), and practice TDD as you build a feature set. What determines your features? Whatever testing skill you’re building. Want to get better at specs for email? Make that to-do list send a project’s tasks to its owner with the click of a button. Don’t wait for a feature request to arise in a production project.

Goodbye, for now

You’ve now got all the tools you need to do basic automated testing in your Rails projects, using RSpec, Factory Girl, Capybara, and DatabaseCleaner to help. These are the core tools I use daily as a Rails developer, and the techniques I’ve presented here show how I learned to effectively use them to increase my trust in my code. I hope I’ve been able to help you get started with these tools as well.

That's the end of *Everyday Rails Testing with RSpec*, but I hope you'll keep me posted as you work toward becoming a test-driven developer. If you have any comments, insights, suggestions, revelations, complaints, or corrections to make to the book, feel free to send them my way:

- Email: aaron@everydayrails.com
- Twitter: https://twitter.com/everydayrails
- Facebook: http://facebook.com/everydayrails
- GitHub: https://github.com/everydayrails/rspec_rails_4/issues

I also hope you'll follow along with new posts at Everyday Rails (http://everydayrails.com/).

Thanks again for reading,

Aaron

More testing resources for Rails

While not exhaustive, the resources listed below have all been reviewed by yours truly and can each play a role in giving you a better overall understanding of Rails application testing.

RSpec

RSpec.info

RSpec.info is the official site for RSpec and provides a couple of basic examples and, more importantly, links to RDocs for all of RSpec's components. You can learn a lot about good use of RSpec through this documentation; take the time to read through it. *http://rspec.info*

RSpec documentation on Relish

Another excellent source for RSpec documentation and examples is the Cucumber-generated output available at Relish. *https://www.relishapp.com/rspec*

Better Specs

Better Specs is a really nice collection of illustrated best practices to employ in your test suite. *http://betterspecs.org*

Peepcode

Great news! The folks at Peepcode have revisited RSpec, replacing their early series of now-outdated videos with the up-to-date *RSpec the Right Way* series. If you'd like to see someone in the act of test-driving development, be sure to check them out. *http://peepcode.com/screencasts/rspec*

The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends

This book, written by RSpec's lead developer, David Chelimsky, provides a thorough look at the entire RSpec ecosystem. I strongly recommend reading this book after you've got the basics down or if you're interested in using RSpec outside of Rails. *http://pragprog.com/book/achbd/the-rspec-book*

Railscasts

I don't know a single Rails developer who's not familiar with Ryan Bates' top-notch screencast series, *Railscasts*. Ryan has done a number of episodes on testing; many either focus on RSpec or include it as part of a larger exercise. Be sure to watch the episode "How I Test," which in part inspired this book. *http://railscasts.com/?tag_id=7*

Code School

Code School's *Testing with RSpec* is a video/hands-on tutorial combination. The course includes content and activities covering configuration, hooks and tags, mocks and stubs, and custom matchers. For a look at Rails' default testing framework, check out *Rails Testing for Zombies*, too. *http://www.codeschool.com/courses/*

The RSpec Google Group

The *RSpec Google Group* is a fairly active mix of release announcements, guidance, and general support for RSpec. This is your best place to go with RSpec-specific questions when you can't find answers on your own. *http://groups.google.com/group/rspec*

Rails testing

Rails Test Prescriptions: Keeping Your Application Healthy

This book by Noel Rappin is my favorite book on Rails testing. Noel does a fine job covering a wide swath of the Rails testing landscape, from Test::Unit to RSpec to Cucumber to client-side JavaScript testing, as well as components and concepts to bring everything together into a cohesive, robust test suite. *http://pragprog.com/book/nrtest/rails-test-prescriptions*

Another good resource from Noel is his talk from Rubyconf 2012, titled *Testing Should Be Fun*. Watch it when you notice your test suite running slow or getting difficult to manage–or better yet, before you've reached that point so you know how to avoid it in the first place. Video available at *http://confreaks.com/videos/1306-rubyconf2012-testing-should-be-fun*; slides at *https://speakerdeck.com/noelrap/testing-should-be-fun*

Rails Tutorial

The book I wish had been around when I was learning Rails, Michael Hartl's *Rails Tutorial*, does the best job of any Rails introduction I've seen of presenting Rails in the way you'll be developing in it–that is, in a test-driven fashion. Also available as a series of screencasts, if that's your learning preference. *http://ruby.railstutorial.org*

Agile Web Development with Rails

Agile Web Development with Rails by Sam Ruby (with Dave Thomas and David Heinemeier-Hansson) *is* the book that *was* available when I got started with Rails. Back in its first edition I thought testing was treated like an afterthought; however, more recent versions do a much better job of weaving tests into the development process. *http://pragprog.com/book/rails4/agile-web-development-with-rails*

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 plugins, gems, and practices 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), 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/. *Everyday Rails Testing with RSpec* is his first book.

Colophon

The cover image of a practical, reliable, red pickup truck[70] is by iStockphoto contributor Habman_18[71]. 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.

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

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

Change log

February 23, 2014

- Replace dropped `end` in "Anatomy of a model spec," chapter 3.
- Use correct title for chapter 4, in book organization section of chapter 1.
- Clarify what I mean by "automating things," chapter 3.
- Attempt to fix syntax highlighting throughout the book. This isn't foolproof and is beyond my control at the moment, but I've done what I can.
- Mention that `before do` is the same as `before :each do`, chapter 3.
- Clarify what I mean by CSS compilation, chapter 9.
- Remove extra `before :each` block in chapter 9's mocking/stubbing example.
- Clarify step of changing the flash message in TDD example, chapter 11.
- Attribute `eq` and `include?` to rspec-expectations, chapter 3.
- Fix path to nested phones, chapter 5.
- Change link to custom matcher examples, chapter 7.

January 24, 2014

- Remove a few remaining references to reqeust specs (now feature specs).
- Other minor typo and language corrections throughout the book.
- Update `selenium-webdriver` version in chapter 8 to address incompatability with latest Firefox.
- Add note about discrepancies between book samples and GitHub project.

January 14, 2014

- Clarify what I mean by "current versions of gems" in chapter 2. I do not update the book or sample application every time a gem is updated. The gems used in the examples were current in summer, 2013.
- Remove unused variable assignment to home_phone is the *phone_spec.rb* example, chapters 3 and 4.
- Change lean syntax example to use build() throughout, instead of create, to match sample project source in chapter 4.
- Fix HTTP verbs in chapter 5.
- Other minor typo and language corrections throughout the book.

October 28, 2013

- Fix minor typo in chapter 1.

October 7, 2013

- Fix typo in chapter 3 (incorrect number of best practices listed).

September 4, 2013

- Clarify issue with spork-rails gem and workaround in Chapter 9.
- Update dependency on selenium-webdriver to 2.35.1 to remove dependency on ruby-zip 1.0.0.

August 27, 2013

- Remove unused examples from previous edition from chapters 5 and 7.
- Add ffaker as alternative to Faker in chapter 4.
- Remove premature reference to factories in chapter 3.

August 21, 2013

- Edited chapters 6-12 and testing resources.
- Switched stub examples to use the new `allow()` syntax.

August 8, 2013

- Edited chapters 1-5.

August 1, 2013

- Updated content for Rails 4.0 and RSpec 2.14.0.
- Replaced chapter 11 with a step-by-step TDD example.

May 15, 2013

- Clarified the state of the sample source for each chapter; each chapter's branch represents the *completed* source.
- Fixed the custom matcher in chapter 7 to properly look for an attribute passed to it.

May 8, 2013

- Corrected reference to `bundle exec rspec` in chapter 2.
- Corrected instructions for grabbing git branches in chapters 3 and 6.
- Fixed Markdown formatting for links to source code and URLs at the very end of the book.

April 15, 2013

- Moved sample code and discussion to GitHub; see chapter 1.
- Updated chapters 9 and 10.
- Reworked the JavaScript/Selenium example in chapter 9.

March 9, 2013

- Fixed stray references to `should` in multiple places.
- Fixed errant model spec for phones in chapter 3.
- Added the changelog to the end of the book.

February 20, 2013

- Fixed formatting error in user feature spec, chapter 8.
- Correctly test for the required `lastname` on a contact, chapter 3.
- Fixed minor typos.

February 13, 2013

- Replaced use of should with the now-preferred expect() syntax throughout most of the book (chapters 9 and 10 excepted; see below).
- Covered the new Capybara 2.0 DSL; chapter 8 now covers feature specs instead of request specs.
- Reworked initial specs from chapter 3 to skip factories and focus on already available methods. Chapter 4 is now dedicated to factories.
- Copy edits throughout.

December 11, 2012

- Added new resources to the resources section.
- Added warnings about the overuse of of Factory Girl's ability to create association data to chapter 4.

November 29, 2012

- Reformatted code samples using Leanpub's improved highlighting tools.
- Added mention of changes in Capybara 2.0 (chapter 8).
- Added warning about using `Timecop.return` to reset the time in specs (chapter 10).

August 3, 2012

- Added the change log back to the book.
- Replaced usage of == to `eq` throughout the book to mirror best practice in RSpec expectations.
- Added clarification that you need to re-clone your development database to test *every* time you make a database change (chapter 3).
- Added a note on the great factory debate of 2012 (chapter 3).
- Added a section about the new RSpec `expect()` syntax (chapter 3).
- Fixed incomplete specs for the #edit method (chapter 5).
- Added an example of testing a non-CRUD method in a controller (chapter 5).
- Added tips on testing non-HTML output (chapter 5).
- Fixed a typo in the `:message` factory (chapter 5).
- Fixed typo in spelling of *transactions* (chapter 8).
- Added a simple technique for testing Rake tasks (chapter 10).

July 3, 2012

- Corrected code for sample factory in chapter 5.

June 1, 2012

- Updated copy throughout the book.
- Added "Testing the Rest" chapter (chapter 10), covering email specs, time-sensitive functionality, testing HTTP services, and file uploads.

May 25, 2012

- Revised chapter 8 on request specs.
- Added chapter 9, covering ways to speed up the testing process and tests themselves.
- Added chapter 11, with tips for becoming a test-driven developer.
- Corrected typos as indicated by readers.

May 18, 2012

- Added chapter 4, which expands coverage on Factory Girl.
- Refactored controller testing into 3 chapters (basic, advanced, cleanup). Advanced includes testing for authentication and authorization.
- Added acknowledgements and colophon.
- Moved resources chapter to an appendix.
- Corrected typos as indicated by readers.

May 11, 2012

- Added sample application code for chapters 1,2, and 3.
- Revised introduction chapter with more information about source code download and purpose.
- Revised setup chapter with changes to generator configuration and Factory Girl system requirements, and other minor changes.
- Revised models chapter to follow along with the sample code better, explain some uses of Factory Girl, and move Faker usage out of chapter (to be added back in chapter 4).
- Switched to using `bundle exec` when calling `rake`, `rspec`, etc. from the command line.
- Added specific gem versions in Gemfile examples.
- Corrected typos as indicated by readers.

May 7, 2012

- Initial release.