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 2019-04-07

Tweet This Book!

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

The suggested tweet for this book is:

I'm learning to test my Rails apps with Everyday Rails Testing with RSpec.

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:

#everydayrailsrspec

Contents

Preface to this edition

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

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

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

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

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

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

[1]https://everydayrails.com

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

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

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

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

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

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

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

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. Thanks also to Ryan Bates for creating Railscasts[3] and teaching me more about Rails than anyone else. The Ruby community just isn't the same without them.

Thanks also 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[4].

Thanks to Andor Chen, Junichi Ito, Toshiharu Akimoto, and Shinkou Gyo, for the work they've done to translate the book to Chinese and Japanese. I'm happy that, through their efforts, I've been able to reach countless more developers than I would have been able to on my own.

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.

[3]http://railscasts.com/
[4]http://www.designojek.com/

1. Introduction

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

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

I've been there. I've never considered myself an engineer in the traditional sense–yet just like traditional engineers, I have problems to solve. And, typically, I find solutions to these problems in building software. I've been developing web applications since 1995. For a long time, I worked as a solo developer on shoestring, public sector projects. Aside from some structured exposure to BASIC as a kid, a little C++ in college, and a wasted week of Java training in my second grown-up job outside of college, I've never had any honest-to-goodness schooling in software development. In fact, it wasn't until 2005, when I'd had enough of hacking ugly spaghetti-style[5] 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.

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

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

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

Why RSpec?

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

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

My second goal with this book is to help you feel comfortable with the RSpec functionality and syntax you're most likely to use on a regular basis. RSpec is a complex framework, but like many complex systems, you'll likely find yourself using 20 percent of the available functionality for 80 percent of your work. With that in mind, this is not a complete guide to RSpec or companion libraries like Capybara, but instead focuses on the tools I've used for years to test my own Rails applications. It will also introduce you to common patterns so that, when you run into an issue that's not covered in the book, you'll be able to intelligently look for solutions and not get stuck.

Who should read this book

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

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

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

Specifically, you should probably have a grasp of

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

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

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

My testing philosophy

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

Discussing the *right* way to test your Rails application can invoke major shouting matches amongst programmers–not quite as bad as, say, the Mac versus PC or Vim versus Emacs debates, but still not something to bring up in an otherwise pleasant conversation with fellow Rubyists. In fact, David Heinemeier-Hansen's keynote at Railsconf 2014, in which he declared TDD as "dead,"[6] recently sparked a fresh round of debates on the topic.

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

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

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

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

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

Yes, there are some tradeoffs–in particular:

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

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

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

How the book is organized

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

The book is organized into the following activities:

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

- In chapter 6, *Testing the User Interface with Feature Specs*, we'll move on to integration testing with feature specs, thus testing how the different parts of our application interact with one another.
- In chapter 7, *Testing the the API with Request Specs*, we'll look at how to test your application's programmatic interface directly, without going through a traditional web user interface.
- In chapter 8, *DRY Specs*, we'll explore when and how to reduce duplication in tests–and when to leave things alone.
- In chapter 9, *Writing Tests Faster, and Writing Faster Tests*, we'll cover techniques for writing efficient, focused tests for fast feedback.
- Chapter 10, *Testing the Rest*, covers testing those parts of our code we haven't covered yet–things like email, file uploads, and external web services.
- I'll go through a step-by-step demonstration of test-driven development in chapter 11, *Toward Test-driven Development*.
- Finally, we'll wrap things up in chapter 12, *Parting Advice*.
- As a bonus, we'll briefly cover the new system spec functionality introduced by RSpec 3.7 in *Appendix A*.

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

Downloading the sample code

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

Get the source!

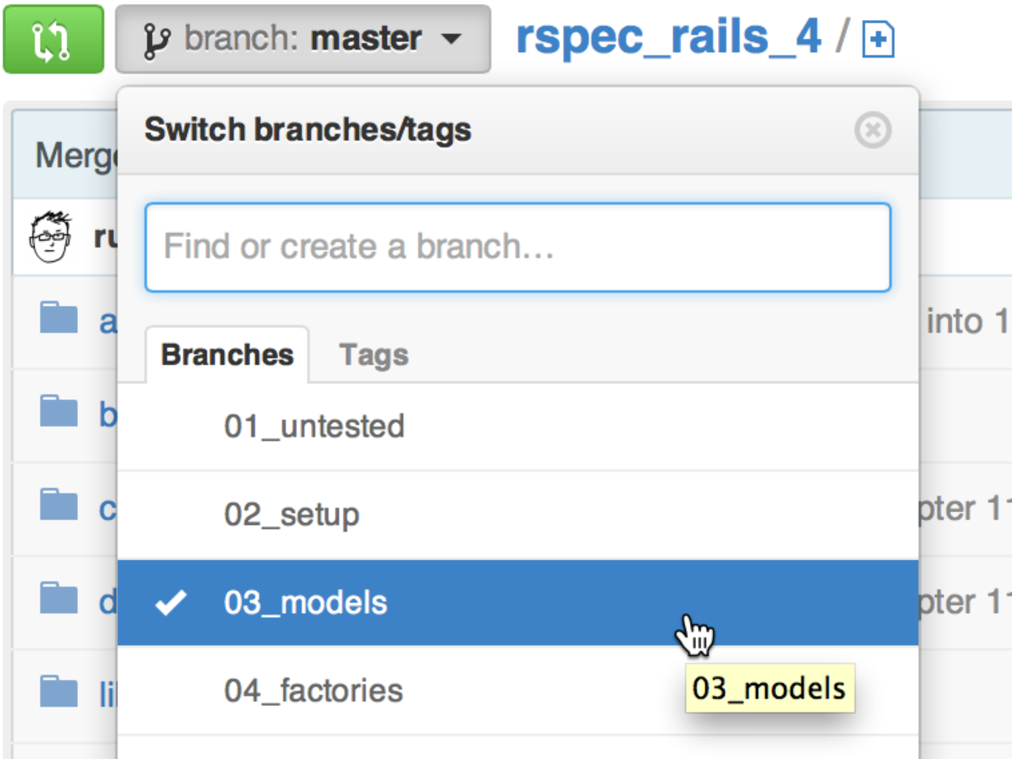
https://github.com/everydayrails/everydayrails-rspec-2017

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

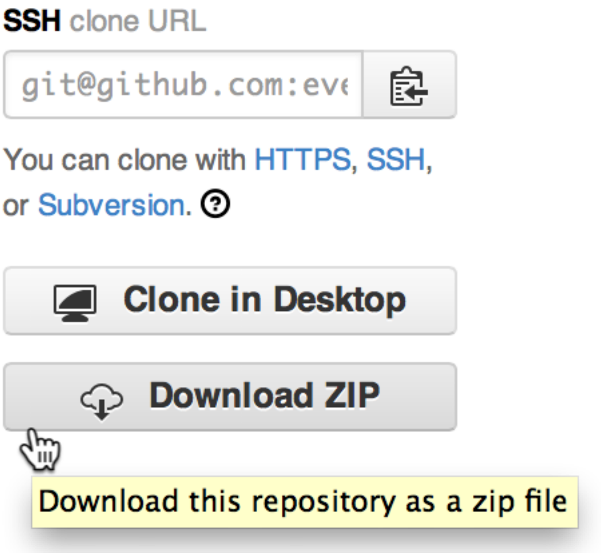
This book is structured so that each chapter builds on the previous one. That way, you can use the previous chapter's branch as a starting point for a current chapter. For example, if I wanted to follow along with chapter 5's code, I could start from chapter 4:

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

If you're not familiar with Git, you may still download the sample code a given chapter. To begin, open the project on GitHub. Then, locate the branch selector and select that chapter's branch:



Finally, click the ZIP download button to save the source to your computer:


SSH clone URL
git@github.com:eve
You can clone with HTTPS, SSH, or Subversion.
Clone in Desktop
Download ZIP
Download this repository as a zip file

Git Immersion[7] is an excellent, hands-on way to learn the basics of Git on the command line. So is Try Git[8].

Code conventions

I'm using the following setup for this application:

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

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

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

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

Again, **this book is not a traditional tutorial!** The code provided here isn't intended to walk you through building an application. It's here to help you understand and learn testing patterns and habits to apply to your own Rails applications. In other words, you can copy and paste, but it's probably not going to do you a lot of good. You may be familiar with this technique from Zed Shaw's Learn Code the Hard Way series[9].

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

Discussion and errata

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

A note about gem versions

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

[9] http://learncodethehardway.org/

About the sample application

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

To start, the application supports the following features:

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

Up to this point, I've been intentionally lazy and only used Rails' default generators to create the entire application (see the *01-untested*[10] branch of the sample code). This means I have a *test* directory full of untouched test files and fixtures. I could run `bin/rails test` at this point, and perhaps some of these tests would even pass. But since this is a book about RSpec, we'll delete this folder, set up Rails to use RSpec instead, and build out a reliable test suite. That's what we'll walk through in this book.

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

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

2. Setting up RSpec

As I mentioned in chapter 1, our project manager application 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 projects, and added and edited data through the web browser. Of course, this approach doesn't scale as we add features. 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, with RSpec at its core. Over the next several chapters, we'll add coverage to the app, starting with RSpec and adding other testing libraries as necessary to round out the suite.

First, we need to install RSpec and configure the application to use it for tests. Once upon a time, it took considerable effort to get RSpec and Rails to work together. That's not 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.
- 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.

You can view all the code changes for this chapter in a single diff[11] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-02-setup origin/01-untested
```

[11] https://github.com/everydayrails/everydayrails-rspec-2017/compare/01-untested...02-setup

Gemfile

Since RSpec isn't included in a default Rails application, we'll need to take a moment to install it. We'll use Bundler to add the dependency. Let's open our *Gemfile* and add RSpec as a dependency:

Gemfile

```ruby
group :development, :test do
  gem 'rspec-rails', '~> 3.6.0'
  # leave other gems provided by Rails
end
```

If you're working with your own, existing application, this might look a little different for you. You'll want to include the *rspec-rails* dependency in such a way that it loads in both the Rails *development* and *test* environments, but not the *production* environment. In other words, you don't want to run tests on your server.

Technically, we're installing the *rspec-rails* library, which includes *rspec-core* and a few other standalone gems. If you were using RSpec to test a Sinatra app, or some other non-Rails Ruby application, you might install these gems individually. rspec-rails packages them together into one convenient installation, along with some Rails-specific conveniences that we'll begin talking about soon.

Run the `bundle` command from your command line to install rspec-rails and its dependencies onto your system. Our application now has the first building block necessary to establish 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:
  <<: *default
  database: db/test.sqlite3
```

Or this if you're using MySQL or PostgreSQL:

config/database.yml

```
test:
  <<: *default
  database: projects_test
```

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

> See the Rails Guide *Configuring Rails Applications*[12] if your database configuration varies from these examples.

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

```
$ bin/rails db:create:all
```

> Prior to Rails 5.0, the previous command would be written as `bin/rake db:create:all`. Versions earlier than 4.1 would replace `bin/rake` with `bundle exec rake`–for example, `bundle exec rake db:create:all`. This book assumes you're using Rails 5.0 or newer, but if not, just replace `bin/rails` with `bin/rake` or `bundle exec rake` when running tasks like creating databases and migrating schema changes. Use `bin/rails` or `bundle exec rails` when using a generator to create new files.

If you didn't yet have a test database, you do now. If you already had one, the `rails` 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.

[12]http://guides.rubyonrails.org/configuring.html

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:

```
$ bin/rails generate rspec:install
```

And the generator reports:

```
Running via Spring preloader in process 28211
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/rails_helper.rb
```

We've now got a configuration file for RSpec (*.rspec*), a directory for our spec files as we create them (`spec`), and two helper files where we'll eventually customize how RSpec will interact with our code (*spec/spec_helper.rb* and *spec/rails_helper.rb*). These last two files include lots of comments to explain what each customization provides. You don't need to read through them right now, but as RSpec becomes a regular part of your Rails toolkit, I strongly recommend reading through them and experimenting with different settings. That's the best way to understand what they do.

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 the *.rspec* file that was just created, and edit it to look like this:

.rspec

```
--require spec_helper
--format documentation
```

Alternatively, you can also add the `--warnings` flag to this file, too. When *warnings* are enabled, RSpec's output will include any and all warnings thrown by your application and gems it uses. This can be useful when developing a real application–always pay attention to deprecation warnings thrown by your tests–but for the purpose of learning to test, I recommend shutting it off and reducing the chatter in your test output. You can always add it back later.

Faster test suite start times with the `rspec` binstub

Next, let's install a binstub for the RSpec test runner, so it can take advantage of faster application boot times via Spring[13]. That way, if you've already booted your app by starting the development server or running a Rake task, the test suite will kick off more quickly since the application is already running. If you don't wish to use Spring for whatever reason, you can skip this section–just remember to use the `bundle exec rspec` command wherever I use `bin/rspec` throughout the book.

Add the following dependency to your *Gemfile*, inside the `:development` group:

Gemfile

```
group :development do
  # Other gems already in this group ...
  gem 'spring-commands-rspec'
end
```

Run `bundle` to install the new dependency, then generate the new binstub:

```
$ bundle exec spring binstub rspec
```

This will create an *rspec* executable, inside the application's *bin* directory.

Try it out!

We don't have any tests yet, but we can still check to see if RSpec is properly installed in the app. Fire it up, using that binstub we just created:

[13]https://github.com/rails/spring

```
$ bin/rspec
```

If everything's installed, you should see output something like:

```
Running via Spring preloader in process 28279
No examples found.

Finished in 0.00074 seconds (files took 0.14443 seconds to load)
0 examples, 0 failures
```

If your output looks different, go back and make sure you've followed the steps outlined above. Don't forget to add the dependencies to your *Gemfile*, *and* run the `bundle` command.

Generators

One more setup step: Telling Rails to generate RSpec-based spec files for us when we use the `rails generate` to add code to our application.

Now that we've got RSpec installed, Rails' stock generators will no longer generate the default MiniTest files in the *test* directory; they'll generate RSpec files in the *spec* directory instead. However, if you'd like, you can manually specify settings for generators. For example, 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 with much depth in this book, so let's narrow down what it creates by default.

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

config/application.rb

```ruby
require_relative 'boot'
require 'rails/all'

Bundler.require(*Rails.groups)

module Projects
  class Application < Rails::Application
    config.load_defaults 5.1

    # comments provided by Rails ...

    config.generators do |g|
      g.test_framework :rspec,
        fixtures: false,
        view_specs: false,
        helper_specs: false,
        routing_specs: false
    end
  end
end
```

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

- `fixtures: false` skips adding files to simplify creating objects in the test database. We'll change this to `true` in chapter 4, when we start using factories to facilitate such data.
- `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.

Boilerplates for model and controller specs will be created by default. If you don't want to automatically generate them, indicate as much in the configuration block. For example, to skip controller specs, you'd add `controller_specs: false`.

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 a helper spec, just add it inside *spec/helpers*, following the spec file naming convention. So if we wanted to test *app/helpers/projects_helper.rb*, we'd add *spec/helpers/projects_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.

Summary

In this chapter, we added RSpec as a dependency to the application's development and test environments, and configured a test-only database for our tests to talk to. We also added some default configuration files for RSpec, as well as configuration for how Rails generators should (or should not) automatically create test files for our application's files.

Now we're ready to write some tests! In the next chapter, we'll start testing 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 `rails test` to verify that there aren't any tests contained within the directory that you may want to port to RSpec.
- **Why don't you test views?** 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 try to relegate testing UI-related code to my integration tests. This is a common practice among Rails developers.

Exercises

If you're working from an existing code base:

- Add *rspec-rails* to your *Gemfile*, and use `bundle` to install. Although this book targets Rails 5.1 and RSpec 3.6, most of the test-specific code and techniques should work with older versions.
- Make sure your application is properly configured to talk to your test database. Create your test database, if necessary.
- Go ahead and configure the Rails `generator` command to use RSpec for any new application code you may add moving forward. You can also just use the default settings provided by *rspec-rails*. This will create extra boilerplate code, which you can delete manually, or ignore. (I recommend deleting unused code.)
- 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 all of these scenarios in the coming chapters.

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

- Follow the instructions for installing RSpec with Bundler.
- Your *database.yml* file may already be configured to use a test database. If you're using a database besides SQLite you may need to create the actual database, if you haven't already, with `bin/rails db:create:all`.
- Optionally, configure Rails' generators to use RSpec, so that as you add new models and controllers to your application, you'll be able to use the generators in your development workflow, and automatically be given starter files for your specs.

Extra credit:

If you create a lot of new Rails applications, you can create a Rails application template[14] to automatically add RSpec and related configuration to your *Gemfile*

[14]http://guides.rubyonrails.org/rails_application_templates.html

and application config files, not to mention create your test database. Daniel Kehoe's excellent Rails Composer[15] is a great starting point for building application templates of your favorite tools.

[15]https://github.com/RailsApps/rails-composer

3. Model specs

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

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

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

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

You can view all the code changes for this chapter in a single diff[16] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-03-models origin/02-setup
```

Anatomy of a model spec

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

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

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

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

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

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

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

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

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

Creating a model spec

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

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

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

RSpec reports that a new file was created:

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

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

spec/models/user_spec.rb

```
require 'rails_helper'

RSpec.describe User, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

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

```
Running via Spring preloader in process 41653

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

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

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


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

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

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

spec/models/user_spec.rb

```ruby
require 'rails_helper'

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

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

```
Running via Spring preloader in process 32556

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

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

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

  2) User is invalid without a first name
     # Not yet implemented
     # ./spec/models/user_spec.rb:5

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

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

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

  6) User returns a user's full name as a string
     # Not yet implemented
```

```
    # ./spec/models/user_spec.rb:9


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

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

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

The RSpec syntax

In 2012, the RSpec team announced a new, preferred alternative to the traditional `should`, added to version 2.11. Of course, this happened just a few days after I released the first complete version of this book–it can be tough to keep up with this stuff sometimes!

This new approach alleviates some technical issues caused by the old `should` syntax[17]. Instead of saying something `should` or `should_not` match expected output, you `expect` something `to` or `not_to` be something else.

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

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

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

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

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

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

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

spec/models/user_spec.rb

```ruby
require 'rails_helper'

RSpec.describe User, type: :model do
  it "is valid with a first name, last name, email, and password" do
    user = User.new(
      first_name: "Aaron",
      last_name:  "Sumner",
      email:      "tester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )
    expect(user).to be_valid
  end

  it "is invalid without a first name"
  it "is invalid without a last name"
  it "is invalid without an email address"
  it "is invalid with a duplicate email address"
  it "returns a user's full name as a string"
end
```

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

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

```
Running via Spring preloader in process 32678

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

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

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

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

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

  4) User is invalid with a duplicate email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:17
  5) User returns a user's full name as a string
     # Not yet implemented
     # ./spec/models/user_spec.rb:18


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

Congratulations, you've written your first test! Now let's get into testing more of our code, so we don't have any more pending tests.

Testing validations

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

spec/models/user_spec.rb

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

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

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

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

spec/models/user_spec.rb

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

And sure enough, RSpec reports a failure:

```
Failures:

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

Finished in 0.06211 seconds (files took 0.28541 seconds to load)
6 examples, 1 failure, 5 pending

Failed examples:

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

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

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

app/models/user.rb

```ruby
class User < ApplicationRecord
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # validates :first_name, presence: true
  validates :last_name, presence: true

  # rest of file omitted ...
```

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

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

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

spec/models/user_spec.rb

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

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

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

spec/models/user_spec.rb

```ruby
it "is invalid with a duplicate email address" do
  User.create(
    first_name:  "Joe",
    last_name:  "Tester",
    email:      "tester@example.com",
    password:   "dottle-nouveau-pavilion-tights-furze",
  )
  user = User.new(
    first_name:  "Jane",
    last_name:  "Tester",
    email:      "tester@example.com",
    password:   "dottle-nouveau-pavilion-tights-furze",
  )
  user.valid?
  expect(user.errors[:email]).to include("has already been taken")
end
```

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

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

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

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

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

spec/models/project_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Project, type: :model do
  it "does not allow duplicate project names per user" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    user.projects.create(
      name: "Test Project",
    )

    new_project = user.projects.build(
      name: "Test Project",
    )

    new_project.valid?
    expect(new_project.errors[:name]).to include("has already been taken")
  end

  it "allows two users to share a project name" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    user.projects.create(
      name: "Test Project",
    )

    other_user = User.create(
      first_name: "Jane",
      last_name:  "Tester",
      email:      "janetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    other_project = other_user.projects.build(
      name: "Test Project",
    )

    expect(other_project).to be_valid
  end
end
```

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

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

app/models/project.rb

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

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

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

Testing instance methods

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

app/models/user.rb

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

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

spec/models/user_spec.rb

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

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

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

Testing class methods and scopes

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

app/models/note.rb

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

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

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  it "returns notes that match the search term" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    project = user.projects.create(
      name: "Test Project",
    )

    note1 = project.notes.create(
      message: "This is the first note.",
      user: user,
    )
    note2 = project.notes.create(
      message: "This is the second note.",
      user: user,
    )
    note3 = project.notes.create(
      message: "First, preheat the oven.",
      user: user,
    )

    expect(Note.search("first")).to include(note1, note3)
    expect(Note.search("first")).to_not include(note2)
  end
end
```

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

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

Testing for failures

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

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  # search results spec ...

  it "returns an empty collection when no results are found" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    project = user.projects.create(
      name: "Test Project",
    )

    note1 = project.notes.create(
      message: "This is the first note.",
      user: user,
    )
    note2 = project.notes.create(
      message: "This is the second note.",
      user: user,
    )
    note3 = project.notes.create(
      message: "First, preheat the oven.",
      user: user,
    )

    expect(Note.search("message")).to be_empty
  end
end
```

This spec checks the value returned by `Note.search("message")`. Since the array *is* empty, the spec passes! We're testing not just for ideal results–the user searches for a term with results–but also for searches with no results.

More about matchers

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

A complete list of RSpec's default matchers may be found in the *README* for the rspec-expectations repository on GitHub[18]. We'll look at several of these throughout this book. In chapter 8, we'll take a look at creating custom matchers of our own.

DRYer specs with describe, context, before and after

So far, the specs we've created for notes have some redundancy: We create the same four objects in each example. Just as in your application code, the DRY principle

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

applies to your tests (with some exceptions, which I'll talk about momentarily). Let's use a few more RSpec features to clean things up.

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

spec/models/note_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Note, type: :model do

  # validation specs

  describe "search message for a term" do
    # searching examples ...
  end
end
```

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

spec/models/note_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Note, type: :model do

  # any other specs

  describe "search message for a term" do

    context "when a match is found" do
      # matching examples ...
    end

    context "when no match is found" 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 or feature; `context` outlines a specific state. In this case, we have a state of a search term with matching results selected, and a state with a non-matching search term selected.

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

spec/models/note_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Note, type: :model do

  before do
    # set up test data for all tests in the file
  end

  # validation tests

  describe "search message for a term" do

    before do
      # set up extra test data for all tests related to search
    end

    context "when a match is found" do
      # matching examples ...
    end

    context "when no match is found" do
      # non-matching examples ...
    end
  end
end
```

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

- `before(:each)` runs before *each* test in a `describe` or `context` block. You can use the alias `before(:example)`, if you prefer, or just `before` as shown in this sample. So if a spec file has four tests in it, the `before` code would run four times.
- `before(:all)` runs once before *all* tests in a `describe` or `context` block. This is aliased as `before(:context)`. This time, the `before` code would run just once, then our four tests would run.
- `before(:suite)` runs before the entire suite of tests, across all files.

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

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

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

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

spec/models/note_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Note, type: :model do
  before do
    @user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    @project = @user.projects.create(
      name: "Test Project",
    )
  end

  it "is valid with a user, project, and message" do
    note = Note.new(
      message: "This is a sample note.",
      user: @user,
      project: @project,
    )
    expect(note).to be_valid
  end

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

  describe "search message for a term" do
    before do
      @note1 = @project.notes.create(
        message: "This is the first note.",
        user: @user,
      )
      @note2 = @project.notes.create(
        message: "This is the second note.",
        user: @user,
      )
      @note3 = @project.notes.create(
        message: "First, preheat the oven.",
        user: @user,
      )
    end

    context "when a match is found" do
      it "returns notes that match the search term" do
        expect(Note.search("first")).to include(@note1, @note3)
      end
    end

    context "when no match is found" do
      it "returns an empty collection" do
        expect(Note.search("message")).to be_empty
      end
    end
  end
end
```

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

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

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

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

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

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

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

How DRY is too DRY?

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

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

That said, well-named variables and methods can also go a long way–for example, in the spec above we used `@note1`, `@note2`, and `@note3` as test notes. But in some cases, you may want to use variables like `@matching_note` or `@note_with_numbers_only`. It depends on what you're testing, but as a general rule, try to be expressive with your variable and method names!

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

Summary

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

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

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

Question

When should I use describe versus context? From RSpec's perspective, you can use `describe` all the time, if you'd like. Like many other aspects of RSpec, `context` exists to make your specs more readable. You could take advantage of this to match a condition, as I've done in this chapter, or some other state[19] in your application.

Exercise

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

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

4. Creating meaningful test data

So far we've been using *plain old Ruby objects* (commonly referred to as *POROs*) to create temporary data for our tests. They are simple and don't require a lot of extra dependencies to work. If they are sufficient for creating meaningful test data for your application, then there's no need to complicate your test suite with anything more.

As testing scenarios become more complex, though, it sure would be nice to keep test data setup simple, even when scenarios are complex, so we can continue to focus 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 a popular gem called *Factory Bot*. 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.
- 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.

You can view all the code changes for this chapter in a single diff[20] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-04-factories origin/03-models
```

[20]https://github.com/everydayrails/everydayrails-rspec-2017/compare/03-models...04-factories

Factories versus fixtures

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

projects.yml

```yaml
death_star:
  name: "Death Star"
  description: "Create the universe's ultimate battle station"
  due_on: 2016-08-29

rogue_one:
  name: "Steal Death Star plans"
  description: "Destroy the Empire's new battle station"
  due_on: 2016-08-29
```

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

A lot of people prefer fixtures. They are relatively fast, and again, since they ship with Rails anyway, they don't add baggage to your application or test suite.

That said, I found fixtures confusing to work with as a beginner. I needed to see the data that were being created in the context of a given test, rather than remember what values were being set in a separate fixture file. In fact, I still prefer this–it's great to have test data setup visible as part of the test, to understand what is happening when the test gets run.

Fixtures can also be brittle, and easily broken, meaning you could spend about as much time maintaining your test data files as you do your tests and application code. Finally, 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. In my opinion, this is not ideal, as it does not represent a real-world situation–if you attempted to create that data from a web form or console, it would fail, right?

For these reasons, when test data setup becomes complex, I reach for **factories**: Simple, flexible, building blocks for test data. If I had to point to a single component that helped me understand testing more than anything else, it would be Factory Bot[21]. It's a little tricky to understand at first, but relatively simple to use once you've got the basics in place.

Prior to October, 2017, Factory Bot was known as Factory Girl.

When used properly (that is, judiciously), factories can keep your tests clean, readable, and realistic. When overused, though, they can lead to slow test runs, as noted by some intelligent, vocal Rubyists[22]. 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, especially for beginners. You can always swap out factories for more efficient approaches later once you've got a suite built and are more comfortable with testing.

Installing Factory Bot

Factory Bot is a new addition to our testing toolkit, so let's add it to the project's `Gemfile`, underneath the existing `rspec-rails` dependency:

Gemfile

```
group :development, :test do
  gem "rspec-rails", "~> 3.6.0"
  gem "factory_bot_rails", "~> 4.10.0"
  # other gems in these groups ...
end
```

Then, run `bundle` from the command line to install it. As the gem's name implies, we're actually installing *factory_bot_rails*, which provides Rails integrations for Factory Bot itself, kind of like how rspec-rails provides extras for using RSpec in Rails. You can use Factory Bot on its own to test non-Rails Ruby code, though that is outside the scope of this book.

[21]https://github.com/thoughtbot/factory_bot
[22]https://groups.google.com/forum/?fromgroups#!topic/rubyonrails-core/_lcjRRgyhC0

While we're at it, let's also tell Rails to automatically create factories for future models, assuming those models get created via a Rails generator. Back in *config/application.rb*, remove the line `fixtures: false,` so the `config.generators` block looks like this:

config/application.rb

```
config.generators do |g|
  g.test_framework :rspec,
    view_specs: false,
    helper_specs: false,
    routing_specs: false,
    request_specs: false
end
```

Adding factories to the application

One of the integrations provided by *factory_bot_rails* is a code generator for building new factories. Let's use it now, to add a User factory:

```
$ bin/rails g factory_bot:model user
```

This adds a new directory inside *spec* called *factories*, and within it, a file called *users.rb* with the following contents:

spec/factories/users.rb

```
FactoryBot.define do
  factory :user do

  end
end
```

Let's fill in the missing information our new factory needs to be useful, borrowing from the test data we created in chapter 3:

spec/factories/users.rb

```ruby
FactoryBot.define do
  factory :user do
    first_name "Aaron"
    last_name  "Sumner"
    email "tester@example.com"
    password "dottle-nouveau-pavilion-tights-furze"
  end
end
```

Now, inside our tests, we can quickly create a new user by calling `FactoryBot.create(:user)`. The user's name will be *Aaron Sumner*. His email address will be *tester@example.com*.

This chunk of code gives us a *factory* we can use throughout our specs. Essentially, whenever we create test data via `FactoryBot.create(:user)`, that user's name will be *Aaron Sumner*, and he'll have an email address and password all ready to go, too.

Although this example's attributes are all strings, you can pass along whatever an attribute's data type expects to see, including integers, booleans, and dates. We'll talk about that more in a few moments.

Let's make sure everything is wired up correctly. Return to the *user_spec.rb* file we set up in the previous chapter and add a quick example to it:

spec/models/user_spec.rb

```ruby
require 'rails_helper'

describe User do
  it "has a valid factory" do
    expect(FactoryBot.build(:user)).to be_valid
  end

  # more specs ...
end
```

Since we're using `FactoryBot.build` here, this instantiates, but does not save, a new user. The user's attributes are assigned by the factory. It then tests that new user's validity, with the `be_valid` matcher introduced in chapter 3. Compare that to this spec from the previous chapter, which created the test data using a PORO:

```
it "is valid with a first name, last name, email, and password" do
  user = User.new(
    first_name: "Aaron",
    last_name:  "Sumner",
    email:      "tester@example.com",
    password:   "dottle-nouveau-pavilion-tights-furze",
  )
  expect(user).to be_valid
end
```

The factory version is more terse, because the details are in a separate file. On the other hand, the PORO version is more self-documenting: We can see what's required to make the test pass.

In reality, there will be good use cases for both approaches. Sometimes, you'll need to see the details. Other times, the details can tucked away. As you gain experience with testing, you should also gain intuition on when to use each technique. Now, though, for the sake of demonstrating our new tool, we'll continue using factories.

Let's revisit the User validation specs from chapter 3. This time, we'll override one or more attributes to generate data from factories, but with specific attributes:

spec/models/user_spec.rb

```
it "is invalid without a first name" do
  user = FactoryBot.build(:user, first_name: nil)
  user.valid?
  expect(user.errors[:first_name]).to include("can't be blank")
end

it "is invalid without a last name" do
  user = FactoryBot.build(:user, last_name: nil)
  user.valid?
  expect(user.errors[:last_name]).to include("can't be blank")
end

it "is invalid without an email address" do
```

```
  user = FactoryBot.build(:user, email: nil)
  user.valid?
  expect(user.errors[:email]).to include("can't be blank")
end
```

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

The next spec is a little different, but uses the same basic tools. This time, we're building a new User with specific values for `first_name` and `last_name`. The test doesn't need to know about `email`, so we accept what the factory gives us. Then, we're making sure that the `name` method on the assigned `user` returns the string we expect.

spec/models/user_spec.rb

```
it "returns a user's full name as a string" do
  user = FactoryBot.build(:user, first_name: "John", last_name: "Doe")
  expect(user.name).to eq "John Doe"
end
```

However, the next example throws in a minor wrinkle:

spec/models/user_spec.rb

```
it "is invalid with a duplicate email address" do
  FactoryBot.create(:user, email: "aaron@example.com")
  user = FactoryBot.build(:user, email: "aaron@example.com")
  user.valid?
  expect(user.errors[:email]).to include("has already been taken")
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 a second `User` persisted in the database–so before running the expectation, we use `FactoryBot.create` to first persist a `user` with the same email address.

Remember: Use `FactoryBot.build` to store a new test object in memory, and use `FactoryBot.create` to persist it in your application's test database.

Generating unique data with sequences

There's something wrong with our current `User` factory. What do you think would happen if we'd written the previous example like this?

spec/models/user_spec.rb

```
it "is invalid with a duplicate email address" do
  FactoryBot.create(:user)
  user = FactoryBot.build(:user)
  user.valid?
  expect(user.errors[:email]).to include("has already been taken")
end
```

Hmm, the test still passes, because the factory always sets a user's email address to *tester@example.com*, unless we tell it to use a different value. This hasn't been an issue with the specs we've written so far, but in cases where we need to add set up multiple users from factories, without directly specifying unique email addresses, we'll get an exception before the actual test code even has a chance to run. For example:

```
it "does something with multiple users" do
  user1 = FactoryBot.create(:user)
  user2 = FactoryBot.create(:user)
  expect(true).to be_truthy
end
```

We get this validation error:

```
Failures:

  1) User does something with multiple users
     Failure/Error: user2 = FactoryBot.create(:user)

     ActiveRecord::RecordInvalid:
       Validation failed: Email has already been taken
```

Factory Bot gives us *sequences* to handle fields with uniqueness validations like these. A sequence increments and injects a counter into the attribute that needs to be unique, each time it creates a new object from a factory. To use one, create the sequence in the factory:

spec/factories/users.rb

```ruby
FactoryBot.define do
  factory :user do
    first_name "Aaron"
    last_name  "Sumner"
    sequence(:email) { |n| "tester#{n}@example.com" }
    password "dottle-nouveau-pavilion-tights-furze"
  end
end
```

See how `n` gets interpolated into the email string? Now, each new user gets a unique, sequential email address–*tester1@example.com*, *tester2@example.com*, and so on.

Associations in factories

Aside from the tricky unique email requirement, our user factory is not too complex–in fact, it could be hard to even justify using a factory instead of a PORO in the cases we've covered so far. But Factory Bot gets more convenient with models that have associations to other models. Let's create factories for notes and projects, using the generator again, to get started (`bin/rails g factory_bot:model note`):

spec/factories/notes.rb

```ruby
FactoryBot.define do
  factory :note do
    message "My important note."
    association :project
    association :user
  end
end
```

And another factory for projects (`bin/rails g factory_bot:model project`):

spec/factories/projects.rb

```ruby
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Project #{n}" }
    description "A test project."
    due_on 1.week.from_now
    association :owner
  end
end
```

Before we continue, let's add a little extra information to the User factory. On the second line, where the factory gets named, give it an alias to *owner* as shown. We'll get into why this is necessary in a moment.

spec/factories/users.rb

```ruby
FactoryBot.define do
  factory :user, aliases: [:owner] do
    first_name "Aaron"
    last_name  "Sumner"
    sequence(:email) { |n| "tester#{n}@example.com" }
    password "dottle-nouveau-pavilion-tights-furze"
  end
end
```

A note belongs to both a project, and a user, but we don't need to manually create one of each in a test–we can just create a note. Check out this example:

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  it "generates associated data from a factory" do
    note = FactoryBot.create(:note)
    puts "This note's project is #{note.project.inspect}"
    puts "This note's user is #{note.user.inspect}"
  end
end
```

Even though we only call Factory Bot once, running the test shows that it's created all the data:

```
Note
This note's project is #<Project id: 1, name: "Test Project 1",
description: "Sample project for testing purposes", due_on:
"2017-01-17", created_at: "2017-01-10 04:01:24", updated_at:
"2017-01-10 04:01:24", user_id: 1>
This note's user is #<User id: 2, email: "tester2@example.com", created_at: "20\
17-01-10 04:01:24", updated_at: "2017-01-10 04:01:24",
first_name: "Aaron", last_name: "Sumner">
```

This example also shows a potential pitfall of using associations in factories, though. Can you spot it? Take a look at the user's email address–why isn't his email address *tester1@example.com*, instead of *tester2@example.com*? It's because the notes factory creates an associated project, which in turn creates an associated user (the associated project's owner), then creates a second user (the associated note's creator).

To work around this, we can update the notes factory, so it defaults to a single user:

spec/factories/notes.rb

```ruby
FactoryBot.define do
  factory :note do
    message "My important note."
    association :project
    user { project.owner }
  end
end
```

Now, the spec results in a single user:

```
Note
This note's project is #<Project id: 1, name: "Test Project 1", description: "S\
ample project for testing purposes", due_on: "2017-01-17", created_at: "2017-01\
-10 04:18:03", updated_at: "2017-01-10 04:18:03", user_id: 1>
This note's user is #<User id: 1, email: "tester1@example.com", created_at: "20\
17-01-10 04:18:03", updated_at: "2017-01-10 04:18:03", first_name: "Aaron", las\
t_name: "Sumner">
```

My point here is that factories can trick you, and generate more test data than you're expecting. It's not a big deal in the contrived test code we've been working with in this section, but it could lead to a confusing debugging session if you're testing something like how many users exist.

Now, let's get back to that *alias* we added to the User factory a moment ago. If you take a look at the *Project* model, you'll see that the User association is called *owner*:

app/models/project.rb

```ruby
class Project < ApplicationRecord
  validates :name, presence: true, uniqueness: { scope: :user_id }

  belongs_to :owner, class_name: User, foreign_key: :user_id
  has_many :tasks
  has_many :notes
end
```

For this to work with Factory Bot, we need to tell the users factory that it may sometimes be referred to as an owner, using an alias. Again, here's the full code for that factory, with the alias in place:

spec/factories/users.rb

```ruby
FactoryBot.define do
  factory :user, aliases: [:owner] do
    first_name "Aaron"
    last_name  "Sumner"
    sequence(:email) { |n| "tester#{n}@example.com" }
    password "dottle-nouveau-pavilion-tights-furze"
  end
end
```

Avoiding duplication in factories

You can define multiple factories that produce the same type of data. For example, to test whether a project is on time or late, we could augment the projects factory by giving each distinct factory a name, an indication of the class it represents, and the attribute or attributes that differentiate it from others (in this case, the `due_on` attribute):

spec/factories/projects.rb

```ruby
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.week.from_now
    association :owner
  end

  factory :project_due_yesterday, class: Project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.day.ago
    association :owner
  end

  factory :project_due_today, class: Project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on Date.current.in_time_zone
    association :owner
  end

  factory :project_due_tomorrow, class: Project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.day.from_now
    association :owner
  end
end
```

Then, we could use those new factories in the Project model's specs. In this case, we'll use a magic matcher–`be_late` isn't built into RSpec, but it's smart enough to assume that there's an attribute or method on a project named `late` or `late?` that returns a boolean–so `be_late` assumes a return value of `true`. Neat, right?

spec/models/project_spec.rb

```ruby
describe "late status" do
  it "is late when the due date is past today" do
    project = FactoryBot.create(:project_due_yesterday)
    expect(project).to be_late
  end

  it "is on time when the due date is today" do
    project = FactoryBot.create(:project_due_today)
    expect(project).to_not be_late
  end

  it "is on time when the due date is in the future" do
    project = FactoryBot.create(:project_due_tomorrow)
    expect(project).to_not be_late
  end
end
```

There's a lot of duplication in those new factories, though. We have to re-define all of a project's attributes every time we define a new factory. That means that, every time we alter the Project model's attributes, we'll need to alter multiple factory definitions.

Factory Bot provides a couple of techniques to reduce duplication. The first is to use **factory inheritance** to only change those attributes that are unique:

spec/factories/projects.rb

```ruby
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.week.from_now
    association :owner

    factory :project_due_yesterday do
      due_on 1.day.ago
    end

    factory :project_due_today do
      due_on Date.current.in_time_zone
    end

    factory :project_due_tomorrow do
      due_on 1.day.from_now
    end
  end
end
```

It may be tricky to see, but with inheritance, the factories for `:project_due_yesterday`, `:project_due_today`, and `:project_due_tomorrow` are nested within the baseline `:project` factory, making the structure like this:

```
factory :project
  factory :project_due_yesterday
  factory :project_due_today
  factory :project_due_tomorrow
```

With inheritance, we can also get rid of the `class: Project` indicator, since this structure lets Factory Bot know to use the Project class in the child factories. We don't need to make any changes to the specs in order for them to pass.

The second technique to help reduce duplication is to compose test data using **traits**. In this approach, we'll modify the factory to define *sets* of attributes. First, let's update the contents of the projects factory:

spec/factories/projects.rb

```ruby
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.week.from_now
    association :owner

    trait :due_yesterday do
      due_on 1.day.ago
    end

    trait :due_today do
      due_on Date.current.in_time_zone
    end

    trait :due_tomorrow do
      due_on 1.day.from_now
    end
  end
end
```

We'll need to alter the spec to use traits. Create a new project with a factory, including the desired trait:

spec/models/project_spec.rb

```ruby
describe "late status" do
  it "is late when the due date is past today" do
    project = FactoryBot.create(:project, :due_yesterday)
    expect(project).to be_late
  end

  it "is on time when the due date is today" do
    project = FactoryBot.create(:project, :due_today)
    expect(project).to_not be_late
  end

  it "is on time when the due date is in the future" do
    project = FactoryBot.create(:project, :due_tomorrow)
    expect(project).to_not be_late
  end
end
```

The real benefit of traits is you can mix and match them to compose complex objects. We will revisit this in later chapters, when our requirements for test data get more complicated.

Callbacks

I want to share one more Factory Bot feature with you. Callbacks let you perform additional actions on a factory-generated object before it is created, or after it is created, built, or stubbed. When used correctly, callbacks can be a powerful timesaver, allowing you to set up complex test scenarios with ease. However, they're also a leading culprit of slow test runs and unnecessarily complicated tests. Use them with care.

Bearing that in mind, let's look at a common use case for callbacks: Creating complex associated data. Factory Bot provides the method `create_list` to facilitate this. Let's add a callback to automatically create notes on a new project. We'll wrap the callback in a trait, so we can limit its use to only where we need it:

spec/factories/projects.rb

```ruby
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.week.from_now
    association :owner

    trait :with_notes do
      after(:create) { |project| create_list(:note, 5, project: project) }
    end

    # Other traits ...
  end
end
```

`create_list` needs the associated model to be created–in this case, we've already got one for notes.

The new `with_notes` trait on the project factory will create a new project, then add five new notes to it, using the notes factory. Now let's look at how to use it in a spec. To start, we'll use a factory with no traits:

spec/models/project_spec.rb

```ruby
it "can have many notes" do
  project = FactoryBot.create(:project)
  expect(project.notes.length).to eq 5
end
```

This will fail, telling us that the project has zero notes, not five:

```
Failures:

  1) Project can have many notes
     Failure/Error: expect(project.notes.length).to eq 5

       expected: 5
            got: 0

       (compared using ==)
     # ./spec/models/project_spec.rb:69:in `block (2 levels) in <top
       (required)>'
     # /Users/asumner/.rvm/gems/ruby-2.4.1/gems/spring-commands-rspec-\
       1.0.4/lib/spring/commands/rspec.rb:18:in `call'
     # -e:1:in `<main>'
```

Let's make it pass by using the new callback we set up in the `with_notes` trait:

spec/models/project_spec.rb

```
it "can have many notes" do
  project = FactoryBot.create(:project, :with_notes)
  expect(project.notes.length).to eq 5
end
```

Now the test will pass, because the callback has created a list of five note objects associated with the project. In the grand scheme of your application, this test isn't very informative, but it does serve as a quick test to make sure the callback is wired up correctly, and gives us a little practice with callbacks before we started creating more complicated tests. In particular, callbacks can be useful for creating test data for Rails models with nested model attributes.

We've only scratched the surface of what you can do with Factory Bot callbacks. Refer to the Factory Bot documentation on callbacks[23] for more uses of this feature. We'll also be using callbacks more later in this book.

[23]https://github.com/thoughtbot/factory_bot/blob/master/GETTING_STARTED.md#callbacks

How to use factories safely

Factories give us a lot of power as test-driven developers. With a few lines of code, we can quickly generate rich sample data to use to validate our software, without encumbering the actual test with a lot of extra setup code.

However, as with any other power tool, I advise a degree of caution when putting factories to work. As already shown in this chapter, use of factories can lead to unexpected data in tests, or unnecessarily slow test runs.

If you see such issues in your tests, first review that your factories are doing only what they need to do, and nothing more. If you need to create associated data with a callback, include that setup in a trait, so it doesn't get triggered every time you use the factory. Use `FactoryBot.build` instead of `FactoryBot.create` whenever possible, to avoid the performance cost of adding data to the test database.

It's also worth asking yourself at the outset, *is a factory even necessary here?* If you can set up your test data using your model's `new` or `create` method, you may be able to avoid a factory altogether. Your tests can also mix and match PORO-generated data and factory-generated data–whatever keeps tests understandable and relatively fast.

> If you've had prior exposure to testing, you may be wondering why I haven't advocated *mocks*, or *doubles*, as a means of speeding up tests. Mocking introduces a whole other layer of complexity. For beginners, I recommend sticking with creating data as outlined so far. We'll introduce mocks, and *stubs*, to our test suite later in the book.

Summary

By using Factory Bot, we've reduced clutter in our specs, and increased our flexibility for creating data to help us test real-life scenarios. The features covered in this chapter are the features I use most often, and should get you through almost all testing tasks. Even with all that we've covered so far, Factory Bot has additional features worth learning more about once you've got these core features mastered, by reading through Factory Bot's documentation[24].

We'll be using Factory Bot throughout the remainder of the book. 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

- Take another look at the model specs we added in chapter 3. Where else could factories help clean them up? Here's one hint: The current coverage in *spec/models/project_spec.rb* has a lot of code for creating User objects. Can you update it to use the users factory, instead?
- Add factories to your application, if you haven't done so already. Remember to use sequences for data that need to be unique.
- Take a look at the factories you created for your app. How can you refactor them with inheritance or traits?
- Try creating some traits with callbacks, and write basic tests to see if they work the way you expected.

[24]https://github.com/thoughtbot/factory_bot/blob/master/GETTING_STARTED.md

5. Controller specs

One of the things I admire about the Rails team is their willingness to remove features that no longer make sense to keep in the framework. In Rails 5.0, two particularly overused testing helpers got the axe, and testing an application's controller layer became officially downplayed–or in proper terms, *soft-deprecated*.

To be honest, I haven't written many controller tests in my own apps in the last few years. Controller tests can easily become brittle, and focused too much about implementation details provided by other parts of your app.

Both the Rails and RSpec teams suggest replacing or removing your app's controller tests (also known as the functional test layer), in favor of directly testing models (units), or with higher-level integration tests. This may sound daunting, but don't worry–it's a change for the better! You already know how to test your application's models, and we'll cover integration testing in the next few chapters.

For now, though, I don't want to completely ignore controller tests, because they still serve a valuable purpose during this transition period. I also think it's important to understand why the Rails team chose to reduce the role of this level of testing. In addition, if you ever pick up a legacy Rails code base with existing RSpec coverage, there's a good chance you'll see some controller specs in there.

We'll explore the basics of controller testing in this chapter:

- First, we'll look at what makes a controller test different from a model test.
- Then we'll test some controller actions.
- Next, we'll look at authenticated actions.
- After that, we'll test user input, including invalid input.
- We'll wrap up with testing a controller method with non-HTML output, such as CSV or JSON.

You can view all the code changes for this chapter in a single diff[25] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

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

Basic controller specs

Let's start with our app's simplest controller. The Home controller has one job: Serve up the application's home page for people who aren't yet signed in.

app/controllers/home_controller.rb

```ruby
class HomeController < ApplicationController

  skip_before_action :authenticate_user!

  def index
  end
end
```

To create tests for the controller, we'll use a generator provided by RSpec:

```
$ bin/rails g rspec:controller home
```

This creates the following boilerplate:

[25]https://github.com/everydayrails/everydayrails-rspec-2017/compare/04-factories...05-controllers

spec/controllers/home_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe HomeController, type: :controller do

end
```

Nothing very interesting so far–in fact, `type: controller` is the only real difference from the model specs we've created to this point. Let's add some simple tests to it. First, we should make sure the controller successfully responds to a browser request. We'll create a `describe` block for the action (`#index`), then add a new spec inside it:

spec/controllers/home_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe HomeController, type: :controller do
  describe "#index" do
    it "responds successfully" do
      get :index
      expect(response).to be_success
    end
  end
end
```

`response` is the object containing all the data your app has to return back to the browser, including the HTTP response code. `be_success` checks whether the response status is successful (a 200 response) or not (for example, a 500 exception).

Let's run it to see what happens. We can focus the test run on just controller tests with `bin/rspec spec/controllers`.

```
Running via Spring preloader in process 44906

HomeController
  #index
    responds successfully

Finished in 0.01775 seconds (files took 0.3492 seconds to load)
1 example, 0 failures
```

Pretty boring, right? That's sort of the point, though. Controller tests should be boring, because *controllers* should be boring. They should take some parameters from the browser, pass them off to Ruby objects that do interesting things with them, and then respond back to the browser. In a perfect world, anyway.

Let's break the test to make sure things are wired up correctly. A simple way to do this is to change `expect(response).to be_success` to expect the response *not* to be successful:

spec/controllers/home_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe HomeController, type: :controller do
  describe "#index" do
    it "responds successfully" do
      get :index
      expect(response).to_not be_success
    end
  end
end
```

Sure enough, it fails.

```
Failures:

1) HomeController#index responds successfully
 Failure/Error: expect(response).to_not be_success
   expected `#<ActionDispatch::TestResponse:0x007f9b16cdd350
   @mon_owner=nil, @mon_count=0,
   @mon_mutex=#<Thread::Mu...:Headers:0x007f9b16cc7230
   @req=#<ActionController::TestRequest:0x007f9b16cdd508 ...>>,
   @variant=[]>>.success?` to return false, got true
```

We could also check for a specific HTTP response code. In this case, we want an actual 200 OK response.

spec/controllers/home_controller_spec.rb

```ruby
it "returns a 200 response" do
  get :index
  expect(response).to have_http_status "200"
end
```

That should pass, too.

Again, these tests appear boring, but there's something secretly interesting about them. Take a look at the controller again, and notice that we're skipping an application-wide `before_action` to authenticate the user before proceeding. What happens if we comment out that line, then run the specs?

```
Failures:

      1) HomeController returns a 200 response
         Failure/Error: get :index

         Devise::MissingWarden:
           Devise could not find the `Warden::Proxy` instance on your
           request environment.
           Make sure that your application is loading Devise and Warden
           as expected and that the `Warden::Manager` middleware is
           present in your middleware stack.
           If you are seeing this on one of your tests, ensure that your
```

```
     tests are either executing the Rails middleware stack or that
     your tests are using the `Devise::Test::ControllerHelpers`
     module to inject the `request.env['warden']` object for you.
```

That's interesting; the tests now fail with a message about missing Devise helpers for our tests. That tells us that the `skip_before_action` line in the controller is doing its job! Later in this chapter, we'll add that support to test controllers that *are* authenticated, but for now, uncomment the line in the home controller to restore functionality.

Authenticated controller specs

Let's create a new controller spec file for the Projects controller, again using the provided generator (`bin/rails g rspec:controller projects`). We can apply the same specs from `home_controller_spec.rb` to the new file:

spec/controllers/projects_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  describe "#index" do
    it "responds successfully" do
      get :index
      expect(response).to be_success
    end

    it "returns a 200 response" do
      get :index
      expect(response).to have_http_status "200"
    end
  end
end
```

Running the specs now, we see that familiar message about missing Devise helpers. Time to do something about them. Devise provides a helper to simulate user login on authenticated controller actions, but we haven't added it. The failure message gives a little more information about how to address the situation. Let's incorporate the helper modules into our test suite. Open *spec/rails_helper.rb* and add the following configuration:

spec/rails_helper.rb

```ruby
RSpec.configure do |config|
  # Other stuff from the config block omitted ...

  # Use Devise test helpers in controller specs
  config.include Devise::Test::ControllerHelpers, type: :controller
end
```

Running the specs still fails, but now the failures provide new information. That means we're making progress. They both fail for the same basic reason–instead of a successful `200` response, we're getting an unsuccessful `302` redirect response. This is because the `index` action requires a user to sign in, but we haven't simulated that in the test.

If you're not using Devise, consult your authentication library's documentation for recommendations on simulating a successful login in a controller spec. If you need to create it yourself. For example, if you're using Rails's built-in `has_secure_password`, you can try creating your own helper method:

```ruby
# roll your own ....
def sign_in(user)
  cookies[:auth_token] = user.auth_token
end
```

Now that we've incorporated the Devise helpers in the test suite, we can set up that simulated login. We need to create a test user, then tell the test to sign in as that user. We'll create the user in a `before` block, so it's applied to both tests, then use the `sign_in` helper to simulate the login:

spec/controllers/projects_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  describe "#index" do
    before do
      @user = FactoryBot.create(:user)
    end

    it "responds successfully" do
      sign_in @user
      get :index
      expect(response).to be_success
    end

    it "returns a 200 response" do
      sign_in @user
      get :index
      expect(response).to have_http_status "200"
    end
  end
end
```

Now, running the specs should pass, because we're accessing the `index` action as an authenticated user.

Let's pause for a moment and think about how we made these tests pass, after we added the necessary supports. We first saw them fail, because we weren't signed in. In the interest of application security, maybe we should test that unauthenticated users–*guests*, if you will–get redirected away from the action. We can expand the test to add this scenario. I like to take advantage of `describe` and `context` blocks to help organize. Let's reconfigure:

spec/controllers/projects_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  describe "#index" do
    context "as an authenticated user" do
      before do
        @user = FactoryBot.create(:user)
      end

      it "responds successfully" do
        sign_in @user
        get :index
        expect(response).to be_success
      end

      it "returns a 200 response" do
        sign_in @user
        get :index
        expect(response).to have_http_status "200"
      end
    end

    context "as a guest" do
      # tests will go here
    end
  end
end
```

Now we've got two contexts within the higher-level `describe` block for the `index` action. The first context covers our authenticated user. Note that the `before` block that creates our test user is nested within that context. Run the specs to make sure everything is in the right place.

Now let's test the case of an unauthenticated user. Alter the `"as a guest"` context to add some coverage:

spec/controllers/projects_controller_spec.rb

```
context "as a guest" do
  it "returns a 302 response" do
    get :index
    expect(response).to have_http_status "302"
  end

  it "redirects to the sign-in page" do
    get :index
    expect(response).to redirect_to "/users/sign_in"
  end
end
```

The first spec should look familiar–it uses the `have_http_status` matcher we used earlier, with the `302` response code we saw reported in the earlier failure. The second spec uses a new matcher, `redirect_to`. In this case, we're checking that the controller intercepts unauthenticated requests, and sends them to the login form provided by Devise.

We can use this same technique to test the app's authorization–can the logged-in user do what they're trying to do? In the controller, this is handled by line 3:

```
before_action :project_owner?, except: [:index, :new, :create]
```

The app requires a user to be a project owner to proceed. Let's add some new coverage, this time to the `show` action. Let's add a new `describe` block and a couple of `context` blocks inside.

spec/controllers/projects_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do

  # index tests here ...

  describe "#show" do
    context "as an authorized user" do
      before do
        @user = FactoryBot.create(:user)
        @project = FactoryBot.create(:project, owner: @user)
      end

      it "responds successfully" do
        sign_in @user
        get :show, params: { id: @project.id }
        expect(response).to be_success
      end
    end

    context "as an unauthorized user" do
      before do
        @user = FactoryBot.create(:user)
        other_user = FactoryBot.create(:user)
        @project = FactoryBot.create(:project, owner: other_user)
      end

      it "redirects to the dashboard" do
        sign_in @user
        get :show, params: { id: @project.id }
        expect(response).to redirect_to root_path
      end
    end
  end
end
```

This time, we create a `@project` for each test. In the first context, the project is owned by the user who's logging in. In the second context, it's owned by a different user.

Another new twist on these tests: We need to pass the project's `id` to the controller action as a `param` value.

Run the tests to see these pass.

Testing user input

We've only used the GET HTTP verb in our tests so far, but of course, users may also interact with the controller via POST, PATCH, or DESTROY requests. Let's add an example of each, starting with a POST. We'll make sure signed-in users can create new projects, and guests are denied access to the action. We've got a context for each:

spec/controllers/projects_controller_spec.rb

```ruby
describe "#create" do
  context "as an authenticated user" do
    before do
      @user = FactoryBot.create(:user)
    end

    it "adds a project" do
      project_params = FactoryBot.attributes_for(:project)
      sign_in @user
      expect {
        post :create, params: { project: project_params }
      }.to change(@user.projects, :count).by(1)
    end
  end

  context "as a guest" do
    it "returns a 302 response" do
      project_params = FactoryBot.attributes_for(:project)
      post :create, params: { project: project_params }
      expect(response).to have_http_status "302"
    end

    it "redirects to the sign-in page" do
      project_params = FactoryBot.attributes_for(:project)
      post :create, params: { project: project_params }
      expect(response).to redirect_to "/users/sign_in"
    end
  end
end
```

The `"as a guest"` context is similar to the tests we wrote earlier for the `index` action, but we're passing params via POST. The expected results for each test, though, are the same.

Here's a look at testing the `update` action. We've got a few scenarios to cover: A user can edit her own project, but not a project owned by somebody else. And guests can't edit any projects. The tests are getting a little more complicated, but they build upon what we've already covered. Let's start with the authorized user specs:

spec/controllers/projects_controller_spec.rb

```
describe "#update" do
  context "as an authorized user" do
    before do
      @user = FactoryBot.create(:user)
      @project = FactoryBot.create(:project, owner: @user)
    end

    it "updates a project" do
      project_params = FactoryBot.attributes_for(:project,
        name: "New Project Name")
      sign_in @user
      patch :update, params: { id: @project.id, project: project_params }
      expect(@project.reload.name).to eq "New Project Name"
    end
  end

  # unauthorized and guest user tests skipped for now ...
end
```

There's one test here–did the attempt to update an existing project succeed? We start with creating a user, and assign a project to that person. Then, in the test itself, we create a new set of project attributes to pass to the action. These represent the values that the user would fill in on the edit project form. `FactoryBot.attributes_for(:project)` creates a hash of new test attributes from the projects factory. We've overridden the factory's default value for `name` with one of our own, to improve testability. Then we simulate user login, and send the new project attributes *along with* the original project's `id` as params, using the PATCH verb. Finally, we check the new value of the `@project` under test, using the `reload` method on it to refresh the value in memory with whatever has been persisted in the database. In this case, the project should have the new name that we passed to the action.

Now let's look at an unauthorized user's access to updating a project:

spec/controllers/projects_controller_spec.rb

```ruby
describe "#update" do
  # authorized user tests omitted ...

  context "as an unauthorized user" do
    before do
      @user = FactoryBot.create(:user)
      other_user = FactoryBot.create(:user)
      @project = FactoryBot.create(:project,
        owner: other_user,
        name: "Same Old Name")
    end

    it "does not update the project" do
      project_params = FactoryBot.attributes_for(:project,
        name: "New Name")
      sign_in @user
      patch :update, params: { id: @project.id, project: project_params }
      expect(@project.reload.name).to eq "Same Old Name"
    end

    it "redirects to the dashboard" do
      project_params = FactoryBot.attributes_for(:project)
      sign_in @user
      patch :update, params: { id: @project.id, project: project_params }
      expect(response).to redirect_to root_path
    end
  end

  # guest tests omitted ...
end
```

These are a little more complicated than the previous examples. This time, we follow the same setup pattern we used when testing an unauthorized user's access to another user's project to set up data. Then, in the actual tests, we first make sure that the project's name doesn't change, and that the unauthorized user is directed to the application dashboard.

After all that, the guest context is fairly simple:

spec/controllers/projects_controller_spec.rb

```ruby
describe "#update" do
  # authorized and unauthorized contexts omitted ...

  context "as a guest" do
    before do
      @project = FactoryBot.create(:project)
    end

    it "returns a 302 response" do
      project_params = FactoryBot.attributes_for(:project)
      patch :update, params: { id: @project.id, project: project_params }
      expect(response).to have_http_status "302"
    end

    it "redirects to the sign-in page" do
      project_params = FactoryBot.attributes_for(:project)
      patch :update, params: { id: @project.id, project: project_params }
      expect(response).to redirect_to "/users/sign_in"
    end
  end
end
```

These may look familiar by now. We're using the same principles covered so far: Create an object, then try to change it through a controller. The attempts fail, instead redirecting the user to a login form.

We’ve got one more action that accepts user input. Let’s turn our attention to the `destroy` action. Similarly to `update`, we want authorized users to be able to delete their own projects, but not those owned by other users. Guests should not be allowed access at all.

spec/controllers/projects_controller_spec.rb

```
describe "#destroy" do
  context "as an authorized user" do
    before do
      @user = FactoryBot.create(:user)
      @project = FactoryBot.create(:project, owner: @user)
    end

    it "deletes a project" do
      sign_in @user
      expect {
        delete :destroy, params: { id: @project.id }
      }.to change(@user.projects, :count).by(-1)
    end
  end

  context "as an unauthorized user" do
    before do
      @user = FactoryBot.create(:user)
      other_user = FactoryBot.create(:user)
      @project = FactoryBot.create(:project, owner: other_user)
    end

    it "does not delete the project" do
      sign_in @user
      expect {
        delete :destroy, params: { id: @project.id }
      }.to_not change(Project, :count)
    end

    it "redirects to the dashboard" do
      sign_in @user
      delete :destroy, params: { id: @project.id }
      expect(response).to redirect_to root_path
    end
  end

  context "as a guest" do
    before do
      @project = FactoryBot.create(:project)
    end

    it "returns a 302 response" do
      delete :destroy, params: { id: @project.id }
      expect(response).to have_http_status "302"
    end

    it "redirects to the sign-in page" do
      delete :destroy, params: { id: @project.id }
      expect(response).to redirect_to "/users/sign_in"
    end

    it "does not delete the project" do
      expect {
        delete :destroy, params: { id: @project.id }
      }.to_not change(Project, :count)
    end
  end
end
```

The only significant new bit to these tests is that we're accessing the `destroy` method via the DELETE verb. As you read through the tests, you should see a lot of patterns that are otherwise familiar at this point in the chapter.

However, if you've played around with our project management application in your browser, you might have noticed a discrepancy between the user interface and the controller–the UI doesn't have a button to let users delete projects! To be honest, this was an accidental omission. It illustrates a shortcoming of relying on controller-level tests, though. If your users can't access a piece of your application's functionality, what's the point? We'll address this shortcoming in subsequent chapters.

Testing user input errors

Look back at the tests we've added for authorized users. So far, we've only covered successful interactions–the user passes valid attributes for a new or edited project, so Rails is able to create or update a record successfully. However, as with model specs, it's often a good idea to verify that things work as planned when something's not right in the controller–in this case, what happens if the validations fail on the new or updated project?

As an example of this, let's modify how we test an authorized user's access to to the `create` action slightly. We'll start by breaking it down into two new contexts: One for valid attributes, and one for invalid attributes. We'll move our existing test into the first context, and add a new test for invalid attributes:

spec/controllers/projects_controller_spec.rb

```
describe "#create" do
  context "as an authenticated user" do
    before do
      @user = FactoryBot.create(:user)
    end

    context "with valid attributes" do
      it "adds a project" do
        project_params = FactoryBot.attributes_for(:project)
        sign_in @user
        expect {
          post :create, params: { project: project_params }
        }.to change(@user.projects, :count).by(1)
      end
    end

    context "with invalid attributes" do
      it "does not add a project" do
        project_params = FactoryBot.attributes_for(:project, :invalid)
        sign_in @user
        expect {
          post :create, params: { project: project_params }
        }.to_not change(@user.projects, :count)
      end
    end
  end

  # other contexts omitted ...
end
```

The new test also uses a new trait on the projects factory, so let's add that, too:

spec/factories/projects.rb

```
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.week.from_now
    association :owner

    # existing traits ...

    trait :invalid do
      name nil
    end
  end
end
```

Now, when we hit the `create` action, we're passing project attributes with no name. The controller won't save the new project in this case.

Handling non-HTML output

Your controllers should have minimal responsibility, but one of the things they *should* be in charge of is making sure data get returned in the proper format. Even though the controller actions we've tested so far all return data in `text/html` format, we haven't concerned ourselves with that in tests.

For a quick demonstration, let's turn to the tasks controller. Since I created it using a Rails scaffold, and haven't made significant changes to the default CRUD actions the scaffold provides, it's technically capable of accepting and responding to requests in either HTML or JSON format. Let's see what happens with a JSON-specific test.

We don't have a spec file for this controller yet, but we can create one quickly with a generator (`bin/rails g rspec:controller tasks`). Then, we can apply what we've learned so far about authentication in controller tests and passing data to controller actions to write a few simple tests.

This test by no means provides comprehensive coverage of the controller's handling of JSON, but it does offer a look at common things you may want to look for in a controller spec file. First, let's take a look at the controller's *show* action:

spec/controllers/tasks_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  before do
    @user = FactoryBot.create(:user)
    @project = FactoryBot.create(:project, owner: @user)
    @task = @project.tasks.create!(name: "Test task")
  end

  describe "#show" do
    it "responds with JSON formatted output" do
      sign_in @user
      get :show, format: :json,
        params: { project_id: @project.id, id: @task.id }
      expect(response.content_type).to eq "application/json"
    end
  end

  # other tests omitted ...
end
```

The setup is really similar to other specs shared earlier in this chapter. We need a user, a project (assigned to the user), and a task (assigned to the project). Then, in the actual test, we sign in as the user, then make a GET request to the controller's show action. The new bit here is that instead of the default HTML format, we need to specify JSON with `format: :json`. This tells the controller to process the request accordingly–which it does, by returning a content type of *application/json*.

To check our work, try expecting `response.content_type` to be *text/html* instead. You should see the test fail accordingly.

Next, let's look at a couple of examples covering the *create* action's JSON handling:

spec/controllers/tasks_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  before do
    @user = FactoryBot.create(:user)
    @project = FactoryBot.create(:project, owner: @user)
    @task = @project.tasks.create!(name: "Test task")
  end

  # `#show` test omitted ...

  describe "#create" do
    it "responds with JSON formatted output" do
      new_task = { name: "New test task" }
      sign_in @user
      post :create, format: :json,
        params: { project_id: @project.id, task: new_task }
      expect(response.content_type).to eq "application/json"
    end

    it "adds a new task to the project" do
      new_task = { name: "New test task" }
      sign_in @user
      expect {
        post :create, format: :json,
          params: { project_id: @project.id, task: new_task }
      }.to change(@project.tasks, :count).by(1)
    end

    it "requires authentication" do
      new_task = { name: "New test task" }
      # Don't sign in this time ...
      expect {
        post :create, format: :json,
          params: { project_id: @project.id, task: new_task }
      }.to_not change(@project.tasks, :count)
      expect(response).to_not be_success
    end
  end
end
```

We'll use the same setup, but this time, we'll send a POST request to the controller's *create* action, and include a `task` parameter like we did earlier in this chapter. Again, we need to tell the spec to send the request in JSON format. From there, we can check–did the request actually make a change to the database? Or, in the event that the user isn't signed in, did it prevent new data in the database?

Summary

We've added a lot of test coverage to our application in this chapter–and we've only touched two of its controllers! Writing controller tests is a quick way to add coverage, but they can also easily grow out of control.

I included several scenarios to illustrate controller tests you might see in real applications, but in my own applications, I try to limit controller tests to making sure access control is wired up correctly. In the case of the tests we've created here, that means tests for unauthorized users and guests. For the things I want authorized users to be able to do, I'll add coverage at higher levels (we'll start on this in the next chapter).

And even if controller testing hasn't been completely removed from Rails or RSpec, it's definitely fallen out of fashion over the years. Remember how we tested the project controller's `destroy` action, only to find it's not even wired to the UI? That's one of several limitations of controller tests.

In short, my advice is to use controller tests when they're the most effective means of testing a piece of functionality, but don't overuse them.

Question

Do I need to test both `success` and `http status`? Not strictly–one or the other should be sufficient, depending on the complexity of your controller's response back to the HTTP client.

Exercises

- We didn't add coverage for the projects controller's `new` or `edit` actions here. Try adding them now. Hint: They'll be similar to the `show` action.
- 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?

6. Testing the user interface with feature specs

So far we've added a good amount of test coverage to our projects software. 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. You may also hear them called *acceptance tests*, or *integration tests*. They provide evidence that your software as a whole is doing what it's supposed to. 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 Bot to generate test data for them. In this chapter, we'll introduce *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.

You can view all the code changes for this chapter in a single diff[26] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-06-features origin/05-controllers
```

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 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. In other words, it tests how all those different units that make up your application integrate with one another.

As of Rails 5.1, support for this type of test is included in the default Rails setup, under the name *system testing*. The concept is generally the same as the feature specs we'll write in this chapter; we'll just be using RSpec instead of the Rails default MiniTest. In this chapter, we'll replace the Rails defaults to use RSpec instead.

Since the time I wrote this chapter, `rspec-rails` has been updated to formally support system specs, as an alternative to feature specs. Appendix A covers the process of converting feature specs to system specs. If you're still learning testing, I recommend working through this chapter and the rest of the book before visiting system specs. The concepts covered here will apply to system specs.

If you're already familiar with testing Rails apps at the browser level, feel free to peek ahead.

[26]https://github.com/everydayrails/everydayrails-rspec-2017/compare/05-controllers...06-features

Additional dependencies

As mentioned earlier, we'll use Capybara to simulate browser interactions–clicking on links, filling in web forms, and checking for expected output on the page. In our Rails 5.1 app, Capybara is already included for us, since it's also a dependency of system tests. If you're working in an older version of Rails, you'll need to add the new dependency. Open the *Gemfile* and create a new `group` for gems that only run in the Rails test environment:

Gemfile

```
group :test do
  gem 'capybara', '~> 2.15.2'
end
```

Unlike the gems we've added so far, Capybara doesn't have code generators that run inside the Rails development environment. We can lighten development's footprint some by making sure Capybara *only* runs in the test environment.

Run the `bundle` command to install Capybara. Next, we need to tell the test suite to load it. Open the `rails_helper` file, and include the library:

spec/rails_helper.rb

```
# This file is copied to spec/ when you run 'rails generate
# rspec:install'
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../../config/environment', __FILE__)
# Prevent database truncation if the environment is production
abort("The Rails environment is running in production mode!") if \
  Rails.env.production?
require 'spec_helper'
require 'rspec/rails'
# Add additional requires below this line. Rails is not loaded
# until this point!
require 'capybara/rspec'

# Rest of RSpec configuration ...
```

Capybara is now ready to use.

A basic feature spec

Capybara lets you write high-level tests using a series of easy-to-understand methods like `click_link`, `fill_in`, and `visit`, to let you describe a usage *scenario* for a feature in your app. Let's write one now. We'll use a generator to place a new test file in the suite. To start, type `rails generate rspec:feature projects` in the command line. Let's look at the file it created:

```ruby
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  pending "add some scenarios (or delete) #{__FILE__}"
end
```

The interesting new thing here is that the new spec file describes, as you might guess, a feature. Next, let's fill it in with a test. Can you guess what this feature spec does, and how?

spec/features/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)

    visit root_path
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"

    expect {
      click_link "New Project"
      fill_in "Name", with: "Test Project"
      fill_in "Description", with: "Trying out Capybara"
      click_button "Create Project"

      expect(page).to have_content "Project was successfully created"
      expect(page).to have_content "Test Project"
      expect(page).to have_content "Owner: #{user.name}"
    }.to change(user.projects, :count).by(1)
  end
end
```

Walking through the steps of this spec, we can see that the spec first creates a new test user, then uses the login form to sign in as that user and create a new project *using the same web form our application's users 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–`home#index`, `sessions#new`, `projects#index`, `projects#new`, `projects#create`, and finally, `projects#show`. However, the results should be the same. A new project is created, the application redirects to the project's page, a flash message is rendered to let us know the process was successful, and the user is listed as the project's owner. All in a single spec!

You may also recognize some techniques from previous chapters–`feature` is used in place of `describe` to structure the spec, and `scenario` describes a given example in place of `it`. We'll come back to `expect{}` in a moment. Inside it, we run through the steps we explicitly want to test in the browser, followed by a series of tests to make sure the resulting view is displayed in a way we'd expect, using the Capybara DSL. It's not quite plain English, but still easy to follow.

At the end of the `expect{}` block, we're using the `change` matcher to ensure one last, important piece: Did the process actually increment the number of projects owned by the user? If so, the test passes.

`click_button` may return before the action it triggers has completed, so it's good practice to have at least one expectation that *will* wait for completion inside an `expect{}` block that depends on it, as this example does.

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.

The Capybara DSL

The test we just added combines RSpec syntax that hopefully is familiar by now (`expect`) with Capybara's methods for doing things in a browser. We `visit` a page, `click_link` to access a hyperlink, `fill_in` form fields `with` values, and `click_button` to process the form input.

There are many more things we can do with Capybara, though. The following example shows some of the other methods provided by the DSL:

```
scenario "works with all kinds of HTML elements" do
  visit "/fake/page"
  click_on "A link or button label"
  check "A checkbox label"
  uncheck "A checkbox label"
  choose "A radio button label"
  select "An option", from: "A select menu"
  attach_file "A file upload label", "/some/file/in/my/test/suite.gif"

  expect(page).to have_css "h2#subheading"
  expect(page).to have_selector "ul li"
  expect(page).to have_current_path "/projects/new"
end
```

You can also *scope* a selector, telling Capybara to use an element specifically `within` another part of the page. For example, suppose this HTML:

```
<div id="node">
  <a href="http://nodejs.org">click here!</a>
</div>
<div id="rails">
  <a href="http://rubyonrails.org">click here!</a>
</div>
```

We can tell the test which *click here!* link to access with:

```
within "#rails" do
  click_link "click here!"
end
```

If Capybara ever complains about ambiguous elements in your tests, try wrapping them in a `within` block to disambiguate.

You can also use Capybara's various `find` methods to locate specific elements and their values. For example:

```
language = find_field("Programming language").value
expect(language).to eq "Ruby"

find("#fine_print").find("#disclaimer").click
find_button("Publish").click
```

These are the Capybara methods I use most often, but it's by no means a complete list of what Capybara adds to your testing toolbox. Read through the Capybara DSL documentation[27] for a full overview, or just keep it handy as a reference. We'll look at a few more of these in future chapters.

Debugging feature specs

Capybara's console output can help track down where tests are failing, but sometimes, its output only tells part of the story. Consider this feature spec. It will fail, because no user is logged in.

[27] https://github.com/teamcapybara/capybara#the-dsl

```ruby
scenario "guest adds a project" do
  visit projects_path
  click_link "New Project"
end
```

The output, though, doesn't necessarily clue us in on the real problem–it just reports that the requested link can't be found on the page:

```
Failures:

  1) Projects guest adds a project
     Failure/Error: click_link "New Project"

     Capybara::ElementNotFound:
       Unable to find link "New Project"
     # rest of stack trace omitted ...
```

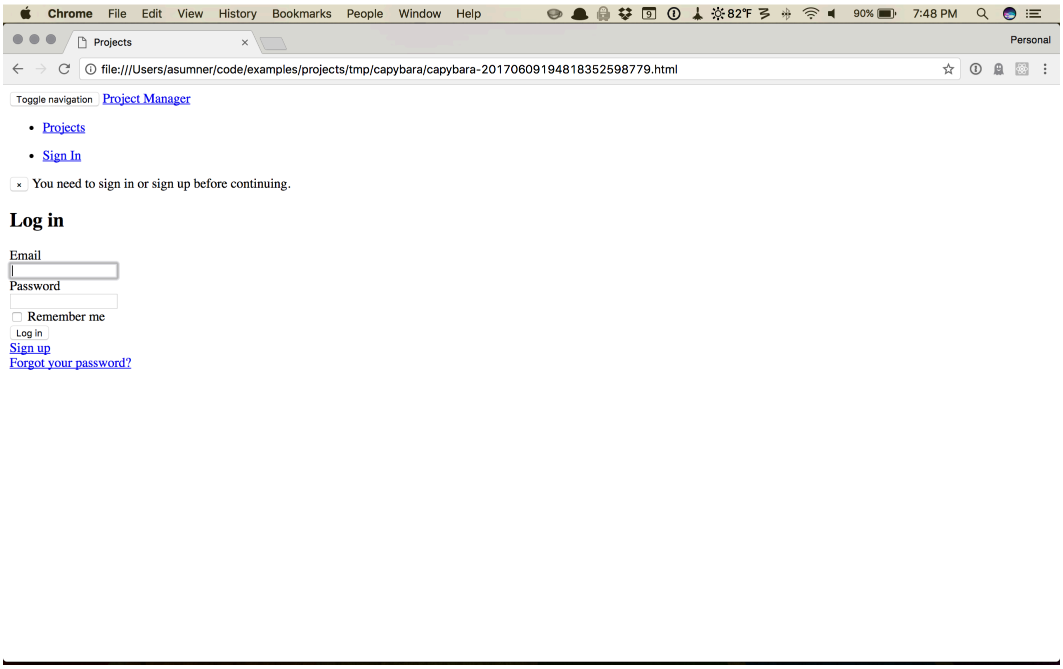
By default, Capybara runs tests using a *headless* browser, so we can't see it go through the steps of the process. We can view the HTML delivered from the Rails application to the browser, though, by inserting `save_and_open_page` immediately before the point of failure:

```ruby
scenario "guest adds a project" do
  visit projects_path
  save_and_open_page
  click_link "New Project"
end
```

Now when we run the test, we still get the same failure, but we get some additional information:

```
File saved to
  /Users/asumner/code/examples/projects/tmp/capybara/
  capybara-201702142134493032685652.html.
Please install the launchy gem to open the file automatically.
  guest adds a project (FAILED - 1)
```

Use the command line or your computer's graphical interface to open the saved file in a browser.


Chrome File Edit View History Bookmarks People Window Help
Projects
Personal
file:///Users/asumner/code/examples/projects/tmp/capybara/capybara-20170609194818352598779.html
Toggle navigation
Project Manager

- Projects
- Sign In

× You need to sign in or sign up before continuing.

Log in

Email

Password

Remember me

Log in

Sign up

Forgot your password?

Aha! We can't access the button, because the user isn't logged in–instead of viewing the projects index, they got re-routed to the sign-in page.

That's helpful, but manually opening the file each time isn't necessary. As the console output implies, this can be automated by installing the *Launchy* gem. Let's add it to the *Gemfile*, then `bundle install`.

Gemfile

```ruby
group :test do
  gem 'capybara', '~> 2.15.2'
  gem 'launchy', '~> 2.4.3'
end
```

Now when we include `save_and_open_page` in a spec, Launchy saves us a step by automatically opening the saved HTML.

`save_and_open_page` is for debugging purposes only; it shouldn't be checked in with your code once the feature is completed. Remove any instances of the method when you don't need them anymore, before committing code to version control.

Testing JavaScript interactions

So we've verified, with a passing spec, that our user interface for adding projects is working as planned. We can use this general approach to test almost all web-based interactions with the application. Unless told otherwise, Capybara will use a simple browser simulator, or driver, to perform the tasks outlined in a test. This driver, called Rack::Test, is fast and reliable, but it doesn't support JavaScript.

Our sample application has one feature that's dependent on JavaScript–if a user clicks the checkbox next to a task, then that task should be marked as completed. Let's write a new spec for that feature. Use the feature spec generator to create a new file for it, or manually add the following to a new file called *spec/features/tasks_spec.rb*:

spec/features/tasks_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Tasks", type: :feature do
  scenario "user toggles a task", js: true do
    user = FactoryBot.create(:user)
    project = FactoryBot.create(:project,
      name: "RSpec tutorial",
      owner: user)
    task = project.tasks.create!(name: "Finish RSpec tutorial")

    visit root_path
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"

    click_link "RSpec tutorial"
    check "Finish RSpec tutorial"

    expect(page).to have_css "label#task_#{task.id}.completed"
    expect(task.reload).to be_completed

    uncheck "Finish RSpec tutorial"

    expect(page).to_not have_css "label#task_#{task.id}.completed"
    expect(task.reload).to_not be_completed
  end
end
```

We're passing the `js: true` option to the scenario here, to let Capybara know to use a JavaScript-capable driver for this particular test. For this application, we'll use the *selenium-webdriver* gem that comes bundled by default with Rails 5.1, and is the default JavaScript driver for Capybara.

By default, Capybara tells selenium-webdriver to use Firefox to run tests, but due to some documented incompatibilities with newer versions of Firefox, we need to configure it to use Chrome instead.

In order to keep the *rails_helper.rb* file relatively clean, let's put this new configuration in a standalone file. RSpec offers support for this; we just need to enable it. Uncomment the following line in RSpec's configuration in *spec/rails_helper.rb*:

spec/rails_helper.rb

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

This lets us put RSpec configuration files in the *spec/support* directory, instead of directly into *spec/rails_helper.rb* like we've done with Devise configurations. Now, create a new file *spec/support/capybara.rb* and add the following configuration to it:

spec/support/capybara.rb

```
Capybara.javascript_driver = :selenium_chrome
```

Finally, for this to all work, we'll need to install *ChromeDriver* to interface with Chrome. ChromeDriver itself is not a Ruby gem, but there *is* a gem to simplify installing it, by installing the *Webdrivers* gem. Let's add it as a dependency now:

Gemfile

```
group :test do
  gem 'capybara', '~> 2.15.4'
  gem 'webdrivers'
  gem 'launchy', '~> 2.4.3'
end
```

Note that Webdrivers includes the *selenium-webdriver* gem as a dependency, so we can remove the explicit dependency in our app. Run `bundle` on your command line to complete the installation.

The first time I used Webdrivers, I ran into an issue installing a version of Chromedriver that supported a dev channel release of Chrome. In the interest of simplicity, I replaced my copy of Chrome with a standard release. Webdrivers then worked out of the box.

If you prefer, you can follow the instructions provided in the official ChromeDriver documentation[a].

[a]https://sites.google.com/a/chromium.org/chromedriver/

Okay, let's give this a try. Run the new spec:

```
$ bin/rspec spec/features/tasks_spec.rb
```

If everything is configured correctly, you'll see a new Chrome window appear (though it may appear behind other windows you have open). In it, our application loads, and invisible fingers click links fill in form fields, and toggles our task as complete and incomplete. Fantastic!

Our new test passes, but look how slowly it runs! This is a downside of tests that require JavaScript to run, and of running them through Selenium. On the other hand, it's been relatively easy to set up so far, and still runs faster than it would take us to manually fill in those fields ourselves. But if a single test takes more than eight seconds (on my computer) to run, imagine how long our site will take as we add more JavaScript functionality and corresponding tests. JavaScript drivers are getting faster, and someday, they may be fast enough to run all specs as fast as Rack::Test does. In the meantime, my advice is to only enable JavaScript on a test when necessary.

Headless drivers

Often, it's not ideal to open a browser window to run a test. For example, when running the suite in a *continuous integration* environment such as Travis CI or Jenkins, the new test will need to run strictly through the command line interface–no new windows allowed. For cases like these, Capybara supports *headless* drivers. We've got a few options to consider, depending on your development and continuous integration environments.

The future-facing solution is to tell Selenium to use Chrome in headless mode when running tests. This is easy to configure, but requires Chrome version 59.0 or higher. After installing that version of Chrome, edit *spec/support/capybara.rb* we need to switch the driver accordingly:

spec/support/capybara.rb

```
Capybara.javascript_driver = :selenium_chrome_headless
```

Now, our JavaScript-dependent tests will run without opening a new browser window.

Another popular headless option, one that I've used for years, is *PhantomJS*. The maintainer of PhantomJS announced he was stepping down from the project upon headless Chrome's pending release, but at the moment it's still a viable option, and is still supported on most hosted continuous integration platforms. If running Chrome (or a version of Chrome that recent) isn't possible for you for whatever reason, then PhantomJS may still be a good alternative.

The PhantomJS installation process will depend on your development environment. On my Mac, I install it via Homebrew with `brew install phantomjs`. You can also download an installer package from the PhantomJS website[28].

To tell RSpec and Capybara to talk to PhantomJS instead of Chrome, we'll need to install the *poltergeist* gem. First, make the following *Gemfile* change, replacing `selenium-webdriver` and `webdrivers` with `poltergeist`:

Gemfile

```ruby
group :test do
  gem 'capybara', '~> 2.15.2'
  gem 'poltergeist'
  gem 'launchy', '~> 2.4.3'
end
```

Run `bundle` to install it, then adjust the Capybara configuration to finish making the switch:

spec/support/capybara.rb

```ruby
require 'capybara/poltergeist'
Capybara.javascript_driver = :poltergeist
```

That's it! Now Capybara will use PhantomJS to run JavaScript tests.

Refer to Capybara's documentation[29] for additional details on setting up these drivers, or others. You may want to experiment with them all to find which will work best for you.

Waiting for JavaScript

By default, Capybara will wait two seconds for the button to appear, before giving up. You can adjust this to any time you'd like:

[28]http://phantomjs.org
[29]https://github.com/jnicklas/capybara

```
Capybara.default_max_wait_time = 15
```

would set the wait time to 15 seconds.

You can include this inside your *spec/support/capybara.rb* file to apply the setting across your entire test suite. (This assumes that you've configured Capybara in your own app as we did in the sample app–it just needs to be someplace that *spec/rails_-helper.rb* can load it.) However, bear in mind that this can cause significant additional slowdown in your test suite. If you need to adjust this setting, it's generally better instead to use `using_wait_time` on a case-by-case basis. For example:

```
scenario "runs a really slow process" do
  using_wait_time(15) do
    # perform test
  end
end
```

As a general rule, do *not* use the Ruby `sleep` method to wait for steps to happen.

Summary

Feature testing builds upon skills you acquired in the previous chapters of this book. It can also be easier to learn and understand, because it's easy to fire up a web browser to understand the steps required to simulate a situation, then recreate those steps using Capybara. This isn't cheating! It's a perfect way to practice your testing skills, and to add coverage to areas of your code that are otherwise untested.

In the next chapter, we'll test how the app interacts with non-human users, and add coverage for its external API.

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 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?

7. Testing the API with request specs

These days, many Rails applications have some sort of external API serving as a backend to a JavaScript-based front-end, native mobile app, or third-party add-ons–either in addition to, or instead of, the type of server-generated user interface we tested in the previous chapter. And since you, the developers consuming your API, and the customers depending on them need it to be reliable, you'll want to test it!

How to write a robust, programmer-friendly public API is beyond the scope of this book, but testing one isn't. The good news is, if you've gone through the chapters on controller testing and feature testing, you've got the basic tools you need to test your API. We'll talk about:

- The differences between request specs and feature specs
- Testing various types of RESTful API requests
- Replacing controller specs with request specs

You can view all the code changes for this chapter in a single diff[30] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-07-requests origin/06-features
```

[30] https://github.com/everydayrails/everydayrails-rspec-2017/compare/06-features...07-requests

Request specs versus feature specs

To begin, where should these tests go? As mentioned in chapter 5, you can test for expected JSON (or XML) output directly in your controller specs. For simple, one-off methods used only by your application, this may suffice. On the other hand, a more robust API calls for integration testing, similar to the feature specs covered in chapter 6. There are a couple of differences, however. With RSpec, the best place for these new, API-specific tests is inside the *spec/requests* directory, separate from the feature specs we've written so far. We also won't use Capybara for these specs. Capybara mimics browser interactions, not programmatic interactions. Instead, we'll use the simple methods we previously used to test controllers' responses to HTTP verbs–`get`, `post`, `delete`, and `patch`.

Our sample application includes a simple API that allows access to a user's list of projects, and for the user to create a new project. Both endpoints require authentication via token. You can see the sample code in *app/controllers/api/projects_controller.rb*. It's not a complex example, but again, this is a book about testing, not designing robust APIs.

Testing GET requests

In the first example, we'll focus on the first endpoint–upon authentication, the client should get back some JSON data containing a list of the user's projects. RSpec provides a generator for request specs, so let's start with that and see what it gives us. On your command line, run:

```
$ bin/rails g rspec:request projects_api
```

Open the new file *spec/requests/projects_apis_spec.rb* and take a look:

spec/requests/projects_apis_spec.rb

```
require 'rails_helper'

RSpec.describe "ProjectsApis", type: :request do
  describe "GET /projects_apis" do
    it "works! (now write some real specs)" do
      get projects_apis_path
      expect(response).to have_http_status(200)
    end
  end
end
```

At first blush, this looks a lot like a controller spec, doesn't it? But as we'll soon see, a request spec can do much more than a controller spec can.

I find it a little awkward that RSpec's generator pluralizes the file name (in this case, *project APIs* versus *project API*), so I usually rename it before continuing. Rename the file to *spec/requests/projects_api_spec.rb*, and add a new test to it:

spec/requests/projects_api_spec.rb

```
require 'rails_helper'

describe 'Projects API', type: :request do
  it 'loads a project' do
    user = FactoryBot.create(:user)
    FactoryBot.create(:project,
      name: "Sample Project")
    FactoryBot.create(:project,
      name: "Second Sample Project",
      owner: user)

    get api_projects_path, params: {
      user_email: user.email,
      user_token: user.authentication_token
    }

    expect(response).to have_http_status(:success)
    json = JSON.parse(response.body)
```

```
      expect(json.length).to eq 1
      project_id = json[0]["id"]

      get api_project_path(project_id), params: {
        user_email: user.email,
        user_token: user.authentication_token
      }

      expect(response).to have_http_status(:success)
      json = JSON.parse(response.body)
      expect(json["name"]).to eq "Second Sample Project"
      # Etc.
    end
  end
```

This example follows some patterns that look less like a controller test, and more like a feature test. The new spec is of type *request*, and starts by creating some sample data. In this case, we've got a user and two projects–one owned by the user, and one that's owned by some other user.

Next, we perform the request via an HTTP GET. As in controller testing, we pass some *params* along to the route. The API requires the user's email address and authentication token to successfully sign in, so those get included in the params. Unlike controller testing, though, we can use *any* route from our app that we'd like–a request spec isn't coupled to a controller, unlike a controller spec. This lets us also make sure the route is pointing where we expect it to.

After that, the test breaks down the returned data, and checks the results. Even though we've got two projects in the database, only one is owned by this user. We can find the ID for that project, then use it to make a second call to the API–this time, to an endpoint that returns more information about a single project. Note that our API requires re-authentication on each call, so we need to pass those params along again.

Finally, we take a look at the JSON returned from this call, and check that the returned project's name matches the one we created in the test data–and it does.

Testing POST requests

The next example sends data to the API:

spec/requests/projects_api_spec.rb

```ruby
require 'rails_helper'

describe 'Projects API', type: :request do

  # First example omitted ...

  it 'creates a project' do
    user = FactoryBot.create(:user)

    project_attributes = FactoryBot.attributes_for(:project)

    expect {
      post api_projects_path, params: {
        user_email: user.email,
        user_token: user.authentication_token,
        project: project_attributes
      }
    }.to change(user.projects, :count).by(1)

    expect(response).to have_http_status(:success)
  end
end
```

Again, we start with some sample data. This time, we need a user and a hash of attributes for a valid project. Then, we check to see if the action we perform makes the expected change–in this case, we want to ensure that the user's total number of projects gets increased by one.

The action is a POST request to the projects API route. We again pass the authentication parameters along, and this time we include those project attributes in the payload. Finally, we check the response status.

Replacing controller specs with request specs

We've focused on testing APIs in these examples, but there's no reason we can't replace the controller specs we wrote in chapter 5 with request specs. Consider the existing coverage of the home controller–it could just as easily be a request spec instead. Create a request spec at *spec/requests/home_spec.rb*, and fill in the new file's contents with:

spec/requests/home_spec.rb

```ruby
require 'rails_helper'

RSpec.describe "Home page", type: :request do
  it "responds successfully" do
    get root_path
    expect(response).to be_success
    expect(response).to have_http_status "200"
  end
end
```

Or a more complex example, such as the coverage of the projects controller's *create* action, could be rewritten as this request spec at *spec/requests/projects_spec.rb* (different from our earlier *projects_api_spec.rb* file):

spec/requests/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.describe "Projects", type: :request do
  context "as an authenticated user" do
    before do
      @user = FactoryBot.create(:user)
    end

    context "with valid attributes" do
      it "adds a project" do
        project_params = FactoryBot.attributes_for(:project)
        sign_in @user
        expect {
          post projects_path, params: { project: project_params }
        }.to change(@user.projects, :count).by(1)
      end
    end

    context "with invalid attributes" do
      it "does not add a project" do
        project_params = FactoryBot.attributes_for(:project, :invalid)
        sign_in @user
        expect {
          post projects_path, params: { project: project_params }
        }.to_not change(@user.projects, :count)
      end
    end
  end
end
```

The differences are subtle–again, instead of coupling directly to the projects controller's `create` action, we can send a POST request to the route instead. Otherwise, we get the same coverage we got in the controller.

Unlike the API controllers, these controller actions use the standard email/password authentication subsystem. So for this to work fully in our project, we'll also need to add some extra configuration to apply Devise's `sign_in` helper to request specs. Let's borrow some code from the Devise project wiki[31] to make it work.

First, create a new file at *spec/support/request_spec_helper.rb*:

[31]https://github.com/plataformatec/devise/wiki/How-To:-sign-in-and-out-a-user-in-Request-type-specs-(specs-tagged-with-type:-:request)

spec/support/request_spec_helper.rb

```ruby
module RequestSpecHelper
  include Warden::Test::Helpers

  def self.included(base)
    base.before(:each) { Warden.test_mode! }
    base.after(:each) { Warden.test_reset! }
  end

  def sign_in(resource)
    login_as(resource, scope: warden_scope(resource))
  end

  def sign_out(resource)
    logout(warden_scope(resource))
  end

  private

  def warden_scope(resource)
    resource.class.name.underscore.to_sym
  end
end
```

Then, in *spec/rails_helper.rb*, tell Devise to apply these helpers to request specs:

spec/rails_helper.rb

```ruby
# Initial setup omitted ...

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

  # Use Devise helpers in tests
  config.include Devise::Test::ControllerHelpers, type: :controller
  config.include RequestSpecHelper, type: :request
end
```

Give the new tests a try by running *bin/rspec spec/requests*. As usual, play around with different expectations. Try to break tests, then make them work again.

So now that you know how to write both controller specs and request specs, which should you write? As I mentioned in chapter 5, I strongly prefer integration specs (features and requests) to controller specs. Controller testing is being phased out in Rails, in favor of these higher-level tests that can cover more area of the application.

How you go about adding coverage to your apps' controller level may vary, though–so for now, I recommend practicing by writing both types of tests, so you can be comfortable with either.

Summary

Understanding how to test your Rails applications' API layers is increasingly important, as we expect apps to talk to each other more and more. Think of your test suite as a client for your API. Even though I said this isn't a chapter about writing APIs, you *can* use your tests to refine the interface you present to other clients.

We've now explored test coverage for every layer of a typical Rails application. However, along the way, we've introduced some repetitive code in our tests. In the next chapter, we'll clean up some of that repetition. We'll also look at why you may *want* to repeat yourself, when it comes to writing tests.

Exercise

- Add another API endpoint to the sample application. It could add to the existing projects API, or provide access to tasks or another feature. Write tests to go along with it. (If you're up to it, try writing the tests first!)
- If you added controller tests to your own application, think now about how to migrate those to request specs. How would they be different?

8. Keeping specs DRY

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, we've got a lot of repetitive code, so technically, we're in violation of the *Don't Repeat Yourself*[32] (DRY) principle.

Just as you would your application code, you should take opportunities to clean up your test suite. In this chapter, we'll address that duplication, using tools provided in RSpec to share code across tests. We'll also look at how a test can be *too* DRY.

In this chapter, we'll discuss:

- Extracting workflows into support modules
- Using `let` as an alternative to reusing instance variables in tests
- Moving common setup into shared contexts
- Writing custom matchers to augment those provided by RSpec and rspec-rails
- Aggregating expectations from multiple specs into a single spec
- Determining what to abstract from a test, and what to keep inline

You can view all the code changes for this chapter in a single diff[33] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-08-dry-specs origin/07-requests
```

Support modules

Let's take a fresh look at our feature specs. We've only written two so far, but they both contain the same steps for logging a user into the app:

[32]http://wiki.c2.com/?DontRepeatYourself
[33]https://github.com/everydayrails/everydayrails-rspec-2017/compare/07-requests...08-dry-specs

```
visit root_path
click_link "Sign in"
fill_in "Email", with: user.email
fill_in "Password", with: user.password
click_button "Log in"
```

If anything changed in the login process–even something as simple as changing the label on the button–we'd have to update it in every place. A simple way to eliminate this duplication is with a **support module**.

Let's start by extracting the code into a new module. Inside the *spec/support* directory, and add a file named *login_support.rb*, with the following contents:

spec/support/login_support.rb

```ruby
module LoginSupport
  def sign_in_as(user)
    visit root_path
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"
  end
end

RSpec.configure do |config|
  config.include LoginSupport
end
```

The module definition includes a single method, made up of the duplicated login steps from the original tests.

Following the module, we open up RSpec configuration to include the new module with `RSpec.configure`. This is optional–you can also explicitly include any support modules you want to use in a given test by including them like this:

spec/features/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  include LoginSupport

  scenario "user creates a new project" do
    # ...
  end
end
```

Now we can simplify the two tests currently using duplicated login steps, and use the new helper method in future tests requiring the same steps. For example, the projects feature spec now looks like this:

spec/features/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    sign_in_as user

    expect {
      click_link "New Project"
      fill_in "Name", with: "Test Project"
      fill_in "Description", with: "Trying out Capybara"
      click_button "Create Project"
    }.to change(user.projects, :count).by(1)

    expect(page).to have_content "Project was successfully created"
    expect(page).to have_content "Test Project"
    expect(page).to have_content "Owner: #{user.name}"
  end
end
```

Extracting common workflows into support modules is one of my favorite ways to reduce code duplication, particularly in feature specs. Make sure you give the methods in these modules names that make their purpose self-explanatory when reading through your tests–if you routinely have to switch to another file to understand a process, you've made your tests a little less useful.

Now that we've done that exercise, guess what? Devise already provides a helper to bypass the login steps entirely, and immediately create a session for a specified user. It's much faster than simulating the UI, and useful when having the user in a logged-in state is a requirement for the important parts of the test. In other words, we're testing project-specific functionality here, not user or login-specific functionality.

To enable it, open up *rails_helper.rb* and add it along with other Devise configurations:

spec/rails_helper.rb

```
RSpec.configure do |config|
  # Other configs ...

  # Use Devise helpers in tests
  config.include Devise::Test::ControllerHelpers, type: :controller
  config.include RequestSpecHelper, type: :request
  config.include Devise::Test::IntegrationHelpers, type: :feature
end
```

Now, replace the call to our custom *sign_in_as* method with a call to the Devise helper, *sign_in*:

spec/features/projects_feature.spec

```
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    sign_in user

    # rest of scenario ...
  end
end
```

Let's run the feature specs and see what happens:

```
Projects
  user creates a new project (FAILED - 1)

Tasks
  user toggles a task (FAILED - 2)

Failures:

  1) Projects user creates a new project
     Failure/Error: click_link "New Project"

     Capybara::ElementNotFound:
       Unable to find link "New Project"
     # stack trace omitted ...

  2) Tasks user toggles a task
     Failure/Error: click_link "RSpec tutorial"

     Capybara::ElementNotFound:
       Unable to find link "RSpec tutorial"
     # stack trace omitted ...
```

Do you see what's going on here? If not, try including a call to `save_and_open_page` right before the failing `click_link` steps in each spec. It looks like our custom login helper, and the original steps we extracted, have a side effect of leaving us on the user home page after login. However, the Devise helper only creates the session, so we have to tell tests where to go to start their workflows:

spec/features/projects_feature.spec

```ruby
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    sign_in user

    visit root_path

    # rest of scenario ...
  end
end
```

Be aware of situations like these going forward. When using something like our custom support method, it's generally a good idea to explicitly add the next step within the test, so the workflow is documented there. Again, for these examples, the fact that a user is signed in is just a setup requirement, not a step to ensure that the core feature under test is working.

We'll use the Devise helper going forward in tests.

Lazy-loading with `let`

We've already been using `before` blocks to DRY up tests, setting up common instance variables before running each test in a `describe` or `context` block. That works well, but has a couple of inherent problems. First, since any code in a `before` will run before *every* test in its corresponding `describe` or `context`, it could affect a test in unexpected ways. If nothing else, it could result in a slower test, if you're creating data you don't actually need. And second, it can lead to readability issues as requirements grow.

To counter these issues, RSpec provides a method, `let`, that *lazy-loads* data only when called upon. `let` also gets called *outside* of a before block, so it requires less structure to set up.

To demonstrate, let's create a model spec for the Task model, since we haven't done that yet. Create the file, either via the `rspec:model` generator or manually creating a file inside *spec/models*. (If you use the generator, it'll also generate a new factory for tasks.) Then, add the following:

spec/models/task_spec.rb

```
 1 require 'rails_helper'
 2 
 3 RSpec.describe Task, type: :model do
 4   let(:project) { FactoryBot.create(:project) }
 5 
 6   it "is valid with a project and name" do
 7     task = Task.new(
 8       project: project,
 9       name: "Test task",
10     )
11     expect(task).to be_valid
12   end
13 
14   it "is invalid without a project" do
15     task = Task.new(project: nil)
16     task.valid?
17     expect(task.errors[:project]).to include("must exist")
18   end
19 
20   it "is invalid without a name" do
21     task = Task.new(name: nil)
22     task.valid?
23     expect(task.errors[:name]).to include("can't be blank")
24   end
25 end
```

This time, the project dependency gets created using `let` on line four, but only in tests that need a project. The first test does: In line eight, `project` calls upon the value set by the `let` in four. `let` creates the new Project, then once the test is done, it gets rid of it after line eleven. The other two tests don't need a project–in fact, the test *is invalid without a project* explicitly does *not* need a project. So in these cases, no project gets created.

One subtle difference–remember how test data set up in `before` blocks need to be stored as instance variables? That's not the case with data created with `let`. So back on line eight, we call upon the data with `project`, not `@project`.

Since `let` creates data on demand, it can get us into trouble if we're not paying attention. Consider this refactor of the Note model's spec, rewritten to use factories and `let`:

spec/models/note_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Note, type: :model do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project, owner: user) }

  it "is valid with a user, project, and message" do
    note = Note.new(
      message: "This is a sample note.",
      user: user,
      project: project,
    )
    expect(note).to be_valid
  end

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

  describe "search message for a term" do
    let(:note1) {
      FactoryBot.create(:note,
        project: project,
        user: user,
        message: "This is the first note.",
      )
    }

    let(:note2) {
      FactoryBot.create(:note,
        project: project,
        user: user,
        message: "This is the second note.",
      )
    }

    let(:note3) {
      FactoryBot.create(:note,
        project: project,
        user: user,
        message: "First, preheat the oven.",
      )
    }

    context "when a match is found" do
      it "returns notes that match the search term" do
        expect(Note.search("first")).to include(note1, note3)
      end
    end

    context "when no match is found" do
      it "returns an empty collection" do
        expect(Note.search("message")).to be_empty
      end
    end
  end
end
```

It's a little cleaner, because we no longer need the `before` blocks to set up data in instance variables. And they pass on the first run! But there's a problem. As an experiment, let's add an extra line to the *returns an empty collection* test:

spec/models/note_spec.rb

```
context "when no match is found" do
  it "returns an empty collection" do
    expect(Note.search("message")).to be_empty
    expect(Note.count).to eq 3
  end
end
```

Then run the spec:

```
Failures:

  1) Note search message for a term when no match is found returns
  an empty collection
     Failure/Error: expect(Note.count).to eq 3

       expected: 3
            got: 0

       (compared using ==)
     # ./spec/models/note_spec.rb:56:in `block (4 levels) in
     <top (required)>'
```

What's going on here? Since we never explicitly called `note1`, `note2`, or `note3` in the test, they never got created, so the search method had no data to search against. So of course the search term found nothing!

We could hack this to work by forcing the notes to load before running the search:

spec/models/note_spec.rb

```
context "when no match is found" do
  it "returns an empty collection" do
    note1
    note2
    note3
    expect(Note.search("message")).to be_empty
    expect(Note.count).to eq 3
  end
end
```

But in my opinion, this *is* a hack. We're striving for readable specs, and the new lines aren't readable. As an alternative, we can use the `let!` variation. Unlike `let`, `let!` doesn't lazy-load. It runs its block right away, so the data inside get created instantly. Replace the calls to `let` with `let!`:

spec/models/note_spec.rb

```
let!(:note1) {
  FactoryBot.create(:note,
    project: project,
    user: user,
    message: "This is the first note.",
  )
}

let!(:note2) {
  FactoryBot.create(:note,
    project: project,
    user: user,
    message: "This is the second note.",
  )
}

let!(:note3) {
  FactoryBot.create(:note,
    project: project,
    user: user,
    message: "First, preheat the oven.",
  )
}
```

And now our experiment passes. But `let!` isn't without its consequences, either. First, we're back to where we started with having test data that's not lazily-loaded. In this case, that's not a big deal: Both of the examples using the test data need all three notes to perform reliably. But do some due diligence to make sure the extra data won't cause unexpected side effects in *all* tests that now have the data as a result.

Second, when reading code, the difference between `let` and `let!` can be easily missed. Again, we're striving for a readable test suite. If you find yourself looking past the subtle difference, consider reverting to `before` blocks and instance variables. There's also nothing wrong with inlining your test data setup directly into the tests that require it[34].

Play around with these different options, and find what works best for you and your team.

Shared contexts

Whereas `let` makes it easy to reuse common test data setup across multiple tests, shared contexts lets you apply that setup across multiple test *files*.

Looking at the tasks controller's spec, the `before` block that runs before each test is a candidate for extraction to a shared context. First, though, maybe we should refactor it to use `let` instead of instance variables. Update the spec to look like the following:

[34]https://robots.thoughtbot.com/my-issues-with-let

spec/controllers/tasks_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project, owner: user) }
  let(:task) { project.tasks.create!(name: "Test task") }

  describe "#show" do
    it "responds with JSON formatted output" do
      sign_in user
      get :show, format: :json,
        params: { project_id: project.id, id: task.id }
      expect(response.content_type).to eq "application/json"
    end
  end

  describe "#create" do
    it "responds with JSON formatted output" do
      new_task = { name: "New test task" }
      sign_in user
      post :create, format: :json,
        params: { project_id: project.id, task: new_task }
      expect(response.content_type).to eq "application/json"
    end

    it "adds a new task to the project" do
      new_task = { name: "New test task" }
      sign_in user
      expect {
        post :create, format: :json,
          params: { project_id: project.id, task: new_task }
      }.to change(project.tasks, :count).by(1)
    end

    it "requires authentication" do
      new_task = { name: "New test task" }
      # Don't sign in this time ...
      expect {
        post :create, format: :json,
          params: { project_id: project.id, task: new_task }
      }.to_not change(project.tasks, :count)
      expect(response).to_not be_success
    end
  end
end
```

The key steps in this first refactor are moving the three lines that were inside the `before` block outside of it, changing the instance variable creation steps to use `let` instead, and then a replacing the instance variables inside the file accordingly. A find-and-replace within the file for each instance variable should do the trick (for example, replace `@project` with `project`).

Run the specs in the file to make sure they still pass. Next, create a new file at *spec/support/contexts/project_setup.rb*, and give it the following contents:

spec/support/contexts/project_setup.rb

```ruby
RSpec.shared_context "project setup" do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project, owner: user) }
  let(:task) { project.tasks.create!(name: "Test task") }
end
```

Finally, back in the controller spec, replace the three `let` lines at the top with a single line:

spec/controllers/tasks_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  include_context "project setup"

  # tests covering show and create ...
```

Run the specs once again; they should still pass. Now, in any spec file needing access to a user, project, or task, we can include the new context with `include_context "project setup"`.

Custom matchers

So far, the matchers provided by RSpec and rspec-rails have covered all of our needs. And there's a very good chance that they always will. However, if your application has a certain condition that you find yourself typing over and over, you may want to create your own, custom matcher.

We have at least one candidate for a custom matcher in our existing tests. In the tasks controller's coverage, we check multiple times that the controller's response is sent with a content type of *application/json*. Since one of RSpec's core tenets is human readability, let's try to improve it with a custom matcher.

Begin by creating the new matcher. Let's add it to *spec/support/matchers/content_-type.rb*. We'll keep things simple to start, and build up to a more fully-featured matcher.

spec/support/matchers/content_type.rb

```
RSpec::Matchers.define :have_content_type do |expected|
  match do |actual|
    content_types = {
      html: "text/html",
      json: "application/json",
    }
    actual.content_type == content_types[expected.to_sym]
  end
end
```

A matcher is defined with a name, which we'll use in our specs to call it–in this case, `have_content_type`. It requires a `match` method and typically works with two values: An *expected* value (what the result to be in order for the matcher to pass), and an *actual* value (what the steps in the test gave us). Here, we *expect* that a shorthand content type (`:html` or `:json`) will have a matching content type value in the `content_types` hash. If it does, then the test passes. If not, then it fails.

Let's try this out. Update the tasks controller to use the new matcher:

spec/controllers/tasks_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  include_context "project setup"

  describe "#show" do
    it "responds with JSON formatted output" do
      sign_in user
      get :show, format: :json,
        params: { project_id: project.id, id: task.id }
      expect(response).to have_content_type :json
    end
  end

  describe "#create" do
    it "responds with JSON formatted output" do
      new_task = { name: "New test task" }
      sign_in user
      post :create, format: :json,
        params: { project_id: project.id, task: new_task }
      expect(response).to have_content_type :json
    end

    # remaining specs ...
  end
end
```

Run the specs; they should still pass. Now let's try to break things! First, try switching `to` to `to_not` on one of the specs. It fails, as expected:

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to_not have_content_type :json
       expected #<ActionDispatch::TestResponse:0x007fc6d2951730
       @mon_owner=nil, @mon_count=0,
       @mon_mutex=#<Thread::Mu...:Headers:0x007fc6d291f4b0
       @req=#<ActionController::TestRequest:0x007fc6d2951960 ...>>,
       @variant=[]>> not to have content type :json
```

That's not very readable. We'll do something about that in a minute. For now, let's try breaking it one other way. Revert the `to_not` back to `to`, and feed it a different content type–for example, `:html` or `:csv`. It will still fail, and still won't be very readable.

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to have_content_type :csv
       expected #<ActionDispatch::TestResponse:0x007fc6d1d353c0
       @mon_owner=nil, @mon_count=0,
       @mon_mutex=#<Thread::Mu...:Headers:0x007fc6d1d0f170
       @req=#<ActionController::TestRequest:0x007fc6d1d356b8 ...>>,
       @variant=[]>> to have content type :csv
```

Let's address that readability now. First, let's extract the hash of content types from the `match` method. RSpec allows for helper methods in matchers to help keep code tidy:

spec/support/matchers/content_type.rb

```
RSpec::Matchers.define :have_content_type do |expected|
  match do |actual|
    begin
      actual.content_type == content_type(expected)
    rescue ArgumentError
      false
    end
  end

  def content_type(type)
    types = {
      html: "text/html",
      json: "application/json",
    }
    types[type.to_sym] || "unknown content type"
  end
end
```

Run tests again, and try breaking them in the same ways. So far, so good–but while the matcher is now a little more readable, its output still is not. We can address that, too: In addition to `match`, RSpec's custom matcher DSL gives us methods for defining the failure message, and the negated failure message–that is, how to report when `to` or `to_not` fails.

spec/support/matchers/content_type.rb

```
RSpec::Matchers.define :have_content_type do |expected|
  match do |actual|
    begin
      actual.content_type == content_type(expected)
    rescue ArgumentError
      false
    end
  end

  failure_message do |actual|
    "Expected \"#{content_type(actual.content_type)} " +
    "(#{actual.content_type})\" to be Content Type " +
    "\"#{content_type(expected)}\" (#{expected})"
  end

  failure_message_when_negated do |actual|
    "Expected \"#{content_type(actual.content_type)} " +
    "(#{actual.content_type})\" to not be Content Type " +
    "\"#{content_type(expected)}\" (#{expected})"
  end

  def content_type(type)
    types = {
      html: "text/html",
      json: "application/json",
    }
    types[type.to_sym] || "unknown content type"
  end
end
```

The tests should still pass. When we make them fail now, though, we get nicer output:

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to_not have_content_type :json
       Expected "unknown content type (application/json)" to
       not be Content Type "application/json" (json)
```

That's a bit nicer. We see what we got as a result (a content type of *application/json*) matches the content type we passed to the matcher, but didn't expect it to (since we deliberately broke it). Back in the spec, revert `to_not` back to `to`, and pass along `:html` instead of `:json`. Run the spec:

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to have_content_type :html
       Expected "unknown content type (application/json)" to
       be Content Type "text/html" (html)
```

Great! We've improved readability. One last thing: `have_content_type` works well, but so might `be_content_type`. We can alias it in the matcher, making the final version of the matcher look like this:

spec/support/matchers/content_type.rb

```ruby
RSpec::Matchers.define :have_content_type do |expected|
  match do |actual|
    begin
      actual.content_type == content_type(expected)
    rescue ArgumentError
      false
    end
  end

  failure_message do |actual|
    "Expected \"#{content_type(actual.content_type)} " +
    "(#{actual.content_type})\" to be Content Type " +
    "\"#{content_type(expected)}\" (#{expected})"
  end

  failure_message_when_negated do |actual|
    "Expected \"#{content_type(actual.content_type)} " +
    "(#{actual.content_type})\" to not be Content Type " +
    "\"#{content_type(expected)}\" (#{expected})"
  end

  def content_type(type)
    types = {
      html: "text/html",
      json: "application/json",
    }
    types[type.to_sym] || "unknown content type"
  end
end

RSpec::Matchers.alias_matcher :be_content_type , :have_content_type
```

So now that we've got a custom matcher, is it a good idea? Arguably, it makes the tests more readable: *Expect response to be content type JSON* is an improvement over *Expect response content type to equal application/json.* But with the new matcher, we now have more code to maintain. Is it worth it? That's ultimately for you and your team to decide, but now you know how to implement a custom matcher, should the need arise.

Before digging too deeply into creating your own matchers, take a look at the *shoulda-matchers* gem. It provides a number of useful matchers to streamline your tests, particularly for model and controller specs, and may already provide a matcher you need. For example, some of the specs we wrote in chapter 3 could be reduced to something as simple as `it { is_expected.to validate_presence_of :name }`.

Aggregating failures

In earlier chapters, I suggested limiting model and controller specs to one expectation each, and allowing feature and request specs to allow as many expectations as necessary to ensure the integration is working. However, in the case of unit tests, this limitation sometimes isn't necessary once coding is complete. And for integration tests, it can often be beneficial to glean the full context of a failure from a test, without having to resort to Launchy (as discussed in chapter 6).

The issue at hand is, as soon as RSpec hits a failing expectation within a test, it stops and reports a failure. Any remaining steps aren't run. But in version 3.3, RSpec gained the ability to **aggregate failures**, so that additional expectations can continue to run and, perhaps, provide some extra context for the failure.

First, let's look at how aggregated failures can tidy up lower-level tests. Back in chapter 5, we created these tests to cover the projects controller:

spec/controllers/projects_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  describe "#index" do
    context "as an authenticated user" do
      before do
        @user = FactoryBot.create(:user)
      end

      it "responds successfully" do
        sign_in @user
        get :index
        expect(response).to be_success
      end

      it "returns a 200 response" do
        sign_in @user
        get :index
        expect(response).to have_http_status "200"
      end
    end

    # rest of specs omitted ...
  end
end
```

These two tests cover really similar ground–as discussed at the time, we could pick one or the other–or we can aggregate them. First, for purposes of demonstration, combine the two tests like this. Be sure to comment out the `sign_in` step (just for a moment).

spec/controllers/projects_controller_spec.rb

```
it "responds successfully" do
  # sign_in @user
  get :index
  expect(response).to be_success
  expect(response).to have_http_status "200"
end
```

Running the spec shows that the first expectation failed, as expected. The second expectation should've failed, too, but it never got a chance to run. Now, let's aggregate the two:

spec/controllers/projects_controller_spec.rb

```
it "responds successfully" do
  # sign_in @user
  get :index
  aggregate_failures do
    expect(response).to be_success
    expect(response).to have_http_status "200"
  end
end
```

And run the specs again:

```
Failures:

  1) ProjectsController#index as an authenticated user responds
  successfully
     Got 2 failures from failure aggregation block.
     # ./spec/controllers/projects_controller_spec.rb:13:in `block (4
     levels) in <top (required)>'
     # /Users/asumner/.rvm/gems/ruby-2.4.1/gems/spring-commands-rspec-
     1.0.4/lib/spring/commands/rspec.rb:18:in `call'
     # -e:1:in `<main>'

     1.1) Failure/Error: expect(response).to be_success
            expected `#<ActionDispatch::TestResponse:0x007faf9dd04dd8
```

```
          @mon_owner=nil, @mon_count=0,
          @mon_mutex=#<Thread::Mu...spatch::Http::Headers:
          0x007faf9b9f5d10
          @req=#<ActionController::TestRequest:0x007faf9dd051e8
          ...>>>>.success?` to return true, got false
        # remaining stack trace omitted ...

   1.2) Failure/Error: expect(response).to have_http_status "200"
          expected the response to have status code 200 but it was 302
        # ./spec/controllers/projects_controller_spec.rb:15:in `block
        (5 levels) in <top (required)>'
        # remaining stack trace omitted ...
```

Great, now both expectations get run, and we can get a little more information about *how* the response was unsuccessful. Uncomment the `sign_in` line, and we should be back to green.

I tend to use aggregated failures more in integration tests. That way, I get the benefit of focused points of failure, without the overhead of running redundant, complex setup across multiple tests. For example, we could aggregate part of the expectations in the project creation scenario, from chapter 6:

spec/features/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    # using our customer login helper:
    # sign_in_as user
    # or the one provided by Devise:
    sign_in user

    visit root_path

    expect {
      click_link "New Project"
      fill_in "Name", with: "Test Project"
      fill_in "Description", with: "Trying out Capybara"
      click_button "Create Project"
    }.to change(user.projects, :count).by(1)

    aggregate_failures do
      expect(page).to have_content "Project was successfully created"
      expect(page).to have_content "Test Project"
      expect(page).to have_content "Owner: #{user.name}"
    end
  end
end
```

That way, if something breaks the flash message, the remaining two expectations still run. One caveat: Aggregated failures only work for *expectations* that fail, not the general conditions required to run the test. In this case, if something broke and the *New Project* link didn't properly render, we'd get an error from Capybara:

```
Failures:

  1) Projects user creates a new project
     Failure/Error: click_link "New Project"

     Capybara::ElementNotFound:
       Unable to find link "New Project"
```

In other words, we're really aggregating *failed expectations*, not general failures. That said, I like the aggregated failures feature a lot, and use it in my tests regularly.

Maintaining test readability

Integration tests can cover a lot of ground. They can test the UI (even with JavaScript interactions), application logic, database activity, and external integrations–sometimes, all within a single test. This can lead to sprawling, hard-to-read tests, requiring context shifts up and down the stack to understand what is happening.

Consider the test we created in chapter 6, covering the task completion interface. In particular, read through the steps we take to mark a task as complete or incomplete, and how we go about testing that the process works as expected:

spec/features/tasks_spec.rb

```ruby
scenario "user toggles a task", js: true do
  # setup and login omitted ...

  check "Finish RSpec tutorial"
  expect(page).to have_css "label#task_#{task.id}.completed"
  expect(task.reload).to be_completed

  uncheck "Finish RSpec tutorial"
  expect(page).to_not have_css "label#task_#{task.id}.completed"
  expect(task.reload).to_not be_completed
end
```

It's not too difficult to read, yet. But what if this feature becomes more complex over time? For example, what if we wanted to track additional details when a task is marked completed, such as the user who marked it as complete, and a timestamp for when it was completed? Each new attribute could require new lines to check the database status and the interface status, along with testing the inverse when the box is unchecked. Our little test is going to start getting long, fast.

We could extract the expectations for each step into standalone helper methods, so when reading through the spec, we're not shifting context. New expectations are then added within the helper methods, rather than inlined directly in the test. This is knowns as *testing at a single level of abstraction*[35]. The idea is to break down a test into well-named methods that provide a high-level view of what's happening inside, without exposing the details in the test itself.

Here's one way to squelch test sprawl in *spec/features/tasks_spec.rb*:

[35]https://robots.thoughtbot.com/acceptance-tests-at-a-single-level-of-abstraction

spec/features/tasks_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Tasks", type: :feature do
  let(:user) { FactoryBot.create(:user) }
  let(:project) {
    FactoryBot.create(:project,
      name: "RSpec tutorial",
      owner: user)
  }
  let!(:task) { project.tasks.create!(name: "Finish RSpec tutorial") }

  scenario "user toggles a task", js: true do
    sign_in user
    go_to_project "RSpec tutorial"

    complete_task "Finish RSpec tutorial"
    expect_complete_task "Finish RSpec tutorial"

    undo_complete_task "Finish RSpec tutorial"
    expect_incomplete_task "Finish RSpec tutorial"
  end

  def go_to_project(name)
    visit root_path
    click_link name
  end

  def complete_task(name)
    check name
  end

  def undo_complete_task(name)
    uncheck name
  end

  def expect_complete_task(name)
    aggregate_failures do
      expect(page).to have_css "label.completed", text: name
      expect(task.reload).to be_completed
    end
  end

  def expect_incomplete_task(name)
    aggregate_failures do
      expect(page).to_not have_css "label.completed", text: name
      expect(task.reload).to_not be_completed
    end
  end
end
```

In the interest of making the actual test more readable, this new version goes through a few rounds of refactoring. First, test data creation is extracted from the test and moved into `let` and `let!` methods, above. Then, each step of the test is extracted into discrete methods: We sign in as the user, then go to the project to test. Finally, we go through the actual steps under test, followed with a custom expectation for each: Complete the task, and confirm that it is indeed completed; undo the completed task, and confirm that it's now incomplete. The test now reads very simply, with the details of *how* we're performing those actions tucked away into separate methods.

Is this better? Personally, I like how readable the new test is–we could share this test with someone who doesn't know how to program, and they could probably understand what is happening from a high-level perspective. I like that, if we change anything about what it means for a task to be complete or incomplete, we can isolate necessary changes to the helper methods. And I like that the new helper methods can be reused across other tests within the file–for example, if we needed scenarios for adding or deleting tasks, we can reuse the new helper methods. (I'll let you create those new tests as an exercise!)

Now, here's what I'm not really happy with in this approach. I don't like that this iteration requires using `let!` to get data ready to use, but that's not necessarily an byproduct of testing at a single level of abstraction. We could continue to refine how test data get created as more scenarios are added to the file. And I've seen this approach to writing tests get overused to the point that I need to dig deeply into a test suite to understand what a helper method is doing, or expecting. In my opinion, that reduces readability while trying to increase it.

I like the concept of testing at a single level of abstraction a lot, but I don't think it's necessary to do until you're comfortable with RSpec and Capybara. Remember, just like your application code, your test suite code can be refined as your skills improve.

Summary

In this chapter, we looked at some approaches you could take to reduce duplication within and across test files. As implied throughout, none of these approaches are *required* to build an effective test suite, but understanding them and applying them judiciously can lead to a test suite that's easy to work with in the long term, just as applying best practices to a Rails code base can keep development pleasant.

Exercise

- Take another look at the current feature spec file for projects. If we added coverage for editing a project, which steps would we reuse from the existing *user creates a new project* scenario? Can you add coverage for this scenario, applying the tools from this chapter to keep the tests DRY, but readable and maintainable?

9. Writing tests faster, and writing faster tests

One of my favorite pieces of programming advice is, "Make it work, make it right, make it fast.[36]" In the first several chapters of this book, we made tests that worked. Then, in chapter 8, we looked at techniques to refactor those tests to reduce duplication. In this chapter, we'll begin to address the *fast* part of the adage.

By *fast*, I mean two things: One, of course, is the amount of time it takes your specs to run. As our app and its test suite grows, running those tests can become excruciatingly slow unless we keep things in check. The goal is to keep the speed reasonable, without sacrificing either the readability afforded us by RSpec, or the peace of mind provided by a robust test suite. 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
- Leveraging your favorite code editor to reduce keystrokes
- Isolating tests from potential performance bottlenecks with mocks and stubs
- Using tags to filter out slow specs
- Techniques for speeding up the suite as a whole

You can view all the code changes for this chapter in a single diff[37] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-09-test-faster origin/08-dry-specs
```

[36] http://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast
[37] https://github.com/everydayrails/everydayrails-rspec-2017/compare/08-dry-specs...09-test-faster

RSpec’s terse syntax

Take a moment to re-read a few of the specs we’ve written so far, particularly some model specs. We’ve been following best practices and providing clear labels for each test, and one expectation per example. It’s all been on purpose. The explicit approach we’ve taken so far mirrors the approach I used when learning to test, and I think it helps to understand what’s going on. However, RSpec provides techniques to continue these best practices while reducing your keystrokes.

In the previous chapter, we looked at `let` as an option for declaring test data. Another method, `subject`, is called similarly, but has a different use case. `subject` lets you declare a test subject, then reuse it implicitly in any number of subsequent examples.

We’ve been using `it` since chapter 3 in its long form, with a human language description of the test in a string. But `it` is just a Ruby method. You may have noticed that the method takes a block, and that block contains the steps of the test. That means for a simple test case, we can potentially reduce things down to a single line!

To accomplish this, we’ll use RSpec’s `is_expected` method. `is_expected` behaves similarly to `expect(something)`, but is intended for one-liner tests.

`specify` is an alias for `it`; some developers prefer `specify` when using RSpec’s terse syntax. 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.

With `is_expected`, we can change a test like

```
it "returns a user's full name as a string" do
  user = FactoryBot.build(:user)
  expect(user.name).to eq "Aaron Sumner"
end
```

to the slightly more succinct

```
subject(:user) { FactoryBot.build(:user) }
it { is_expected.to satisfy { |user| user.name == "Aaron Sumner" } }
```

We get the same results, and can reuse the `subject` implicitly on subsequent tests, without having to retype it for each example.

That said, I don't tend to use this terse syntax throughout my test suites, nor do I use it in cases where the `subject` would only apply to a single test. I *do* like this syntax, though, when used in conjunction with **Shoulda Matchers**. Shoulda was initially a standalone testing framework–a complete alternative to RSpec. While it's no longer in development, its extensive collection of matchers for ActiveModel, ActiveRecord, and ActionController were extracted into a standalone gem that can be used in RSpec (or MiniTest). By including one additional gem, we can reduce some of our specs from three or four or five lines down to one or two.

To use Shoulda Matchers, we first need to add the dependency. Add it along with other testing-related gems in the *Gemfile*. Since we're using Rails 5.1 for this project, we'll need to use a Rails 5.x-compatible branch. If you're using an older version of Rails, you can require the gem without the extra `git` and `branch` options.

Gemfile

```
group :test do
gem 'capybara', '~> 2.15.4'
gem 'selenium-webdriver'
gem 'launchy', '~> 2.4.3'
gem 'shoulda-matchers',
  git: 'https://github.com/thoughtbot/shoulda-matchers.git',
  branch: 'rails-5'
end
```

After running `bundle` from the command line, we need to configure the test suite to use the new dependency. Open *spec/rails_helper.rb* and include the following at the bottom of the file, indicating that we intend to use it with RSpec and Rails:

spec/rails_helper.rb

```
Shoulda::Matchers.configure do |config|
  config.integrate do |with|
    with.test_framework :rspec
    with.library :rails
  end
end
```

Now we can start shortening some specs. Let's start with the four validation tests in our User model specs. We can reduce them down to four lines!

spec/models/user_spec.rb

```
it { is_expected.to validate_presence_of :first_name }
it { is_expected.to validate_presence_of :last_name }
it { is_expected.to validate_presence_of :email }
it { is_expected.to validate_uniqueness_of(:email).case_insensitive }
```

We're using two matchers provided by Shoulda Matchers–`validate_presence_of` and `validate_uniqueness_of`–to write these tests. Since the uniqueness validation on `email` is set by Devise, we need to let the spec know that the validation isn't case sensitive–thus, `case_insensitive`.

Next, let's turn our attention to our Project model's spec–specifically, the tests we wrote to ensure that a user couldn't have more than one project with the same name, but two different users can have projects sharing a name. With Shoulda Matchers, we can reduce these down to a single, one-line test:

spec/models/project_spec.rb

```
it { is_expected.to validate_uniqueness_of(:name).scoped_to(:user_id) }
```

I really like this style of testing at the model layer, particularly when doing test-first development on a model. For example, given a new model called *widget*, I'll open a spec and think about how a widget should look like or act, and write my tests:

```
it { is_expected.to validate_presence_of :name }
it { is_expected.to have_many :dials }
it { is_expected.to belong_to :compartment }
it { is_expected.to validate_uniqueness_of :serial_number }
```

Then I'll write the code to make them pass. This is a great way to think about the code you need to write, and make sure your application code meets its requirements.

As I write this, the Shoulda Matchers project is in need of a new maintainer. I still use it regularly, but if you're a more conservative developer, you may wish to stick with longer tests. Or maybe this is a chance for you to contribute to open source!

Editor shortcuts

A critical part of learning to program is understanding the ins and outs of your preferred code editor. I personally use Atom[38] for most of my coding (and writing), and have learned just enough of its syntax for creating snippets[39] to reduce much of the routine typing for my tests. For example, typing `desc`, followed by a tab, in my editor creates a `describe...end` block, and places my cursor inside it to add code at the right place.

If you also use Atom, you can install the Everyday Rails RSpec Atom package[40] to try my shortcuts for yourself–refer to the documentation for the full set of snippets. If you use a different editor, take some time to learn the shortcuts it provides you, as well as how to create shortcuts of your own. Less time typing boilerplate code means more time providing business value!

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

[38]https://atom.io
[39]http://flight-manual.atom.io/using-atom/sections/snippets/
[40]https://atom.io/packages/atom-everydayrails-rspec

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*. The mock *stands in* for an object we may previously have used a factory or PORO to create. However, 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.

RSpec includes a rich mocking library. In some projects you may see other libraries used, such as Mocha. Factory Bot, which we've been using since chapter four to create fake data for our tests, includes a method for building a stubbed object, too. In this section, we'll focus on RSpec's built-in support.

Let's look at a few examples. In the Note model, we've got a line that creates an attribute `user_name` on a given note, by way of `delegate`. Given what we know so far, we could test how that works like this:

spec/models/note_spec.rb

```
it "delegates name to the user who created it" do
  user = FactoryBot.create(:user, first_name: "Fake", last_name: "User")
  note = Note.new(user: user)
  expect(note.user_name).to eq "Fake User"
end
```

In this example, we need to persist a User object, just so we can access its `first_name` and `last_name` attributes in the test. It's only a fraction of a second here, but those fractions of seconds can build up over time–especially in more complex setup scenarios. Mocking is commonly used to reduce access to the database in cases like this.

Also, this test is about the Note model, but it has to know an awful lot about how the User model is implemented–why should it care that a user's `name` is calculated from its `first_name` and `last_name`? All it really needs to know is that an associated user will return a string called `name`.

Here's a modified version of the test, using a *mock* User object, and a *stubbed* method on the Note under test:

spec/models/note_spec.rb

```
it "delegates name to the user who created it" do
  user = double("user", name: "Fake User")
  note = Note.new
  allow(note).to receive(:user).and_return(user)
  expect(note.user_name).to eq "Fake User"
end
```

Here, we replace the persisted User object with a test double. The double isn't a real User–in fact, if you were to inspect it, you'd see its class is `Double`. The double only knows how to respond to requests on `name`. To illustrate, try adding an expectation to the test:

spec/models/note_spec.rb

```
it "delegates name to the user who created it" do
  user = double("user", name: "Fake User")
  note = Note.new
  allow(note).to receive(:user).and_return(user)
  expect(note.user_name).to eq "Fake User"
  expect(note.user.first_name).to eq "Fake"
end
```

This would pass in the original example (before switching to a mock), but fails here:

```
1) Note delegates name to the user who created it
   Failure/Error: expect(note.user.first_name).to eq "Fake"
     #<Double "user"> received unexpected message :first_name with
      (no args)
```

The double *only* knows to respond to `name`, because that's all the note needs for the code to work. Go ahead and remove the extra `expect`.

Now, let's look at the stub, created here with `allow`. This line tells the test runner that, at some point within the scope of the test, our code is going to call `note.user`. When that happens, instead of finding the value of `note.user_id`, looking up the value in the database for that User, and returning the found User object, just return the Double called `user` instead.

As a result, we've got a test that's isolated away from the implementation details that reside outside of the model under test, and removes two database calls–we no longer persist a user, and we no longer do a lookup on the user.

A common, and valid, critique of this approach is that, if we were to rename or remove the `User#name` method, the new test would continue to pass. Try it for yourself now–comment out the `User#name` method, and run the test.

A basic RSpec double doesn't verify that its stubbed methods exist on the object it's standing in for. To protect against this, we can use a *verified double* instead. Let's verify that the double behaves like an instance of a User (notice the capitalized `User` now, required for the verified double to work):

spec/models/note_spec.rb

```ruby
it "delegates name to the user who created it" do
  user = instance_double("User", name: "Fake User")
  note = Note.new
  allow(note).to receive(:user).and_return(user)
  expect(note.user_name).to eq "Fake User"
end
```

Now, if something changes to that `name` method, we'll get a failure:

```
1) Note delegates name to the user who created it
     Failure/Error: user = instance_double("User", name: "Fake User")
       the User class does not implement the instance method: name.
       Perhaps you meant to use `class_double` instead?
```

In this case, we didn't mean to use `class_double`–we can just uncomment the `name` method to restore the test.

As implied by the failure message, we can also verify a class method on a double, using `class_double`.

When using doubles to mock out objects for your tests, use verified doubles whenever possible. This will help protect you from erroneously passing tests.

If you're working in a code base with an existing suite of RSpec tests, or have followed other testing tutorials, you may notice mocks and stubs used heavily in controller tests. In fact, I've seen some developers go to extremes in attempts to avoid touching the database in controller tests (and I'll admit, I've done this, too).

Here's an example that isolates testing a controller method completely from the database. The `index` action on our notes controller deviates from the original `index` provided by the generator, in that it's calling the `search` scope on the Note model to gather results to return back to the browser. Let's test this, but keep the database out of the mix. Here we go:

spec/controllers/notes_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe NotesController, type: :controller do
  let(:user) { double("user") }
  let(:project) { instance_double("Project", owner: user, id: "123") }

  before do
    allow(request.env["warden"]).to receive(:authenticate!).and_return(user)
    allow(controller).to receive(:current_user).and_return(user)
    allow(Project).to receive(:find).with("123").and_return(project)
  end

  describe "#index" do
    it "searches notes by the provided keyword" do
      expect(project).to receive_message_chain(:notes, :search).
        with("rotate tires")
      get :index,
        params: { project_id: project.id, term: "rotate tires" }
    end
  end
end
```

Let's walk through this. First, we use `let` to lazily assign a `user` and `project` for the test (we covered `let` in chapter 8). Since we don't call any methods on the mock user, we can safely use a regular double in this case. `project`, on the other hand, has attributes for `owner` and `id` that we'll use, so it's safer to use a verified double for it.

Next, in the `before` block, we begin by stubbing out the `authenticate!` and `current_user` methods that Devise relies on, since this is a password-protected page in the app. We also stub out `Project.find`, as provided by Active Record, to return the mock `project` instead of attempting a lookup in the database. As long as `Project.find(123)` gets called somewhere in the code under tests, we'll get the `project` double in lieu of an actual project object.

Finally, we need to verify that our controller code does what it's supposed to do–in this case, call the `search` scope on the `notes` association of the `project`, with a search `term` that matches a param by the same name. We do all that with the lines

spec/controllers/notes_controller_spec.rb

```
expect(project).to receive_message_chain(:notes, :search).
  with("rotate tires")
```

Here, `receive_message_chain` lets us refer to `project.notes.search`. Something to remember–in order to pass, this expectation needs to be added *before* the code that actually exercises your application's code. Using `expect` instead of `allow` here causes a test failure if the message chain is *not* received.

Play around with this code a bit, as well as the controller code it tests. Find ways to break it, and observe the output provided by RSpec.

Now, let's consider the practicality of this new test code. Is it fast? It's a lot faster than it would be if we had to create a user and a project in the database, log in as the user, and then call the controller action with the required params. If our suite had a lot of tests like this, minimizing calls to the database would keep the suite from growing too slow.

On the other hand, this code is kind of tricky to read. Some of the setup is boilerplate that could be extracted, such as the Devise authentication stubbing. But thinking about object setup in a different way than you'd normally create and save objects in your application code can add a lot of cognitive overhead to testing, especially for beginners.

We're also ignoring a commonly-held rule of mocking: "Don't mock what you don't own.[41]" In other words, we've mocked out two pieces of functionality provided by third party libraries, not our own code–the authentication layer provided by Devise, and Active Record's `find` method in the interest of speed and isolation. Since we don't own the third party libraries, and we've mocked their behavior, our tests may not alert us when changes to those libraries break application code.

That said, there are times when it makes sense to mock things your application doesn't own. For example, if you need to make a slow network call, or interact with a rate-limited external API, mocking can save you from the expense. As your experience with testing and Rails grows, you may find it makes sense to create your own code that interacts directly with those interfaces, then stub out *that* code where it gets used in the application–but keep a set of tests you can run on the new code directly against the expensive interface. That is beyond the scope of this book, or our little test application.

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, anyway. 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.

[41]https://8thlight.com/blog/eric-smith/2011/10/27/thats-not-yours.html

Tags

Suppose we're working on a new feature in our app. Its test coverage includes individual tests on a couple of models, a controller, and an integration test. We'd rather not run the *entire* suite while developing the new feature, but it's also some hassle to run each individually. This is where RSpec's tag option[42] helps. With tags, you can flag specific tests to run, skipping everything else.

It's common to use a tag called `focus` to accomplish this. Add the tag to the test or tests you want to run:

```
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:

```
$ bin/rspec --tag focus
```

If you want to *skip* a tag–for example, you don't want to run a particularly slow integration test right now, you can tag the slow tests appropriately, then invert the tag with a tilde:

```
$ bin/rspec --tag ~slow
```

This runs every test *except* those tagged as `slow`.

Tags may be applied on `describe` and `context` blocks, too, meaning the tag will affect any test within those blocks.

Don't forget to remove focus tags before committing your new code, and always run the entire suite without the tags before determining that a new feature is complete. On the other hand, you may find it useful to keep tags like `slow` in the code base for the long term.

[42] https://relishapp.com/rspec/rspec-core/docs/command-line/tag-option

If you find yourself using the focus tag regularly, you can configure RSpec to assume that if it finds *any* tests with the tag, then to use it. If no examples are found with the tag, then run the whole suite.

spec/spec_helper.rb

```
RSpec.configure.do |config|
  config.filter_run focus: true
  config.run_all_when_everything_filtered = true
  # other configs ...
end
```

You can also configure RSpec to always skip examples with specific tags; for example:

spec/spec_helper.rb

```
RSpec.configure do |config|
  config.filter_run_excluding slow: true
  # other configs ...
end
```

You can still run the tagged `slow` tests explicitly, from the command line:

```
$ bin/rspec --tag slow
```

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, but don't need to run it on a regular basis, mark it as a spec to skip:

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

I recommend this over commenting out the test–since skipped 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.

Earlier versions of RSpec provided this functionality via the `pending` method. `pending` is still around, but is quite different: Now, any spec marked pending will still be run. If it fails, its result is shown as pending. However, if it passes, it'll be marked as a failure.

Run tests in parallel

I have seen a test suite's runtime grow to 30 minutes, and I've heard plenty of stories of even slower suites. A good way to get an already slow suite in check is by running its tests in parallel, using the ParallelTests gem[43]. With ParallelTests, your suite is divided among multiple cores, for potentially large speed boosts. Running that 30 minute suite through six cores brought its runtime down to about seven minutes. It's a huge boost, especially when you don't have time to address individual slow tests.

I like this approach, but be careful–it can sometimes mask other questionable testing habits, such as overusing slow, expensive integration tests. Don't rely solely on ParallelTests to speed up your suite–use it in conjunction with other techniques.

Take Rails out of the equation

Incorporating mocks and tags will both 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. Using Spring, via binstubs, to decrease boot time helps, as we've done throughout this book.

[43]https://github.com/grosser/parallel_tests

But if you *really* want to speed up your test suite, you can go all out and remove Rails from the equation entirely. Whereas Spring still loads the framework–but limits itself to loading once–these solutions go one step further by not loading the framework at all.

This is a lot 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[44] and the Destroy All Software[45] screencast series from Gary Bernhardt.

Summary

Up until this chapter, we didn't talk about a lot of alternatives 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 3, or in a more terse fashion as shared here. You can also choose from mocks and stubs, or factories, or basic Ruby objects, or any combination thereof, to load and work with test data. Finally, you now know about options for loading and running your test suite.

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–including a first look at the test-first development process. Before that, though, 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 the terse syntax we explored in this chapter. 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.)

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

- 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[46] to learn more about *how* it implements those matchers.
- Look for opportunities to experiment with mocking, even if you don't keep the code. Start small, with focused tests in your models, and work your way to more complex scenarios. Do they make the tests run more quickly? Are the new tests still easy to understand?
- Using RSpec tags, identify your slow specs. Run your test suite including and excluding the tests. What kind of performance gains do you see?

[46]https://github.com/thoughtbot/shoulda-matchers

10. Testing the rest

We're building solid test coverage across the entire project management application now. We've tested our models and controllers, and also tested them in tandem with views via feature specs. Our external API also has coverage, via request specs. But we're missing a few things–what about those file uploads on the Note model, outgoing email, and the geocoding integration? We can test these facets, too!

In this chapter we'll explore:

- Testing file uploads.
- Testing background jobs performed by Active Job.
- How to test for email delivery.
- Testing against external web services.

You can view all the code changes for this chapter in a single diff[47] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-10-testing-the-rest origin/09-test-faster
```

Testing file uploads

Uploading attached files is such a common requirement of web applications that there are currently at least three actively-maintained libraries to simplify adding file uploads (or you can create your own). But how to test? Our sample application uses Paperclip, so we'll focus on testing with it, but the general approach applies to other libraries. Let's start with a feature spec to cover notes:

[47] https://github.com/everydayrails/everydayrails-rspec-2017/compare/09-test-faster...10-testing-the-rest

spec/features/notes_spec.rb

```
require 'rails_helper'

RSpec.feature "Notes", type: :feature do
  let(:user) { FactoryBot.create(:user) }
  let(:project) {
    FactoryBot.create(:project,
      name: "RSpec tutorial",
      owner: user)
  }

  scenario "user uploads an attachment" do
    sign_in user
    visit project_path(project)
    click_link "Add Note"
    fill_in "Message", with: "My book cover"
    attach_file "Attachment", "#{Rails.root}/spec/files/attachment.jpg"
    click_button "Create Note"
    expect(page).to have_content "Note was successfully created"
    expect(page).to have_content "My book cover"
    expect(page).to have_content "attachment.jpg (image/jpeg"
  end
end
```

This is really similar to other feature specs we've written, but uses Capybara's `attach_file` method to simulate the file attachment process. The first argument indicates the form field label, and the second is a path to a test file's location. Here, we've got a new directory called *spec/files*, and it contains a small JPEG file that will be used in the test. You can call this directory just about anything. I've seen *spec/fixtures* used for this purpose, for example. Whatever you name the file, remember to check it into your version control, since it will be required for other developers to successfully run the test suite.

Once the test file is put in place, the new spec should pass as written, but there's a problem. We're using Paperclip's default configuration and storing uploads to the file system, but the same path is used in both the Rails *development* and *test* environments–meaning there's a chance that the test suite will overwrite files you may have uploaded when clicking through the app's interface in development mode. I'll admit, this has happened to me!

To work around this, Paperclip (and other file upload libraries) can be overridden with an alternative upload path. Let's follow the approach outlined in Paperclip's documentation, and amend the configuration for our app's test environment:

config/environments/test.rb

```
# Keep files uploaded in tests from polluting the Rails development
# environment's file uploads
Paperclip::Attachment.default_options[:path] = \
  "#{Rails.root}/spec/test_uploads/:class/:id_partition/:style.:extension"
```

Run the new test again, and observe where Paperclip stores the file–it should now be in the *spec/test_uploads* directory. That's fine, but it would be nice to clean up old uploads once we're done running tests. Again, let's borrow from Paperclip's documentation and keep things tidy:

spec/rails_helper.rb

```
RSpec.configure do |config|
  # other configs in this block omitted ...

  # Clean up file uploads when test suite is finished
  config.after(:suite) do
    FileUtils.rm_rf(Dir["#{Rails.root}/spec/test_uploads/"])
  end
end
```

Now, once tests are finished, RSpec will wipe out the directory and its contents. Let's make sure it doesn't accidentally wind up in source control by including it in the project's *.gitignore*.

.gitignore

```
# Ignore uploads from Paperclip
/public/system
/spec/test_uploads
```

Let's review what we just did, but ignore Paperclip for a moment. First, we created a spec with the file upload step, and provided a file for it to use. Next, we specified an alternate upload path for use in testing only. Finally, we told RSpec to clean up after the suite is finished. These are the three basic steps to getting file uploads under test at a feature level. If you're not using Paperclip, consult documentation for your upload library of choice to see how to apply these steps to your own application. If you've got your own Ruby in place to handle file uploads, you may be able to stub out the upload directory using `allow` (see chapter 9) or by extracting the upload path's location to an environment-dependent configuration value.

Since we introduced Shoulda Matchers in the previous chapter, we can take advantage of built-in support provided by Paperclip in model-level specs. To enable this support, we need to make a couple more config changes in *spec/rails_helper.rb*. First, require the matchers near the top of the file:

spec/rails_helper.rb

```
require 'spec_helper'
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../../config/environment', __FILE__)
abort("The Rails environment is running in production mode!") \
  if Rails.env.production?
require 'rspec/rails'
require 'capybara/rspec'
require 'paperclip/matchers'

# rest of file omitted ...
```

Then include the module within the `RSpec.configure` block:

spec/rails_helper.rb

```ruby
RSpec.configure do |config|
  # earlier configuration omitted ...

  # Add support for Paperclip's Shoulda matchers
  config.include Paperclip::Shoulda::Matchers
end
```

Now, we can test that the model supports file uploads with a one-line test:

spec/models/note_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Note, type: :model do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project, owner: user) }

  it { is_expected.to have_attached_file(:attachment) }

  # Other tests omitted ...
```

Our implementation is basic, but if your Paperclip integration is more complex, refer to the Paperclip documentation[48] for additional matchers. Other file upload libraries have similar support–consult their respective documentation for more details.

Finally, if you find yourself repeatedly attaching files in tests, you may want to consider including the attachment as an attribute when using the model in a test. For example, we could do this with our Note factory by adding a new trait, as introduced in chapter 4:

[48]http://www.rubydoc.info/gems/paperclip/Paperclip/Shoulda/Matchers

spec/factories/notes.rb

```ruby
FactoryBot.define do
  factory :note do
    message "My important note."
    association :project
    user { project.owner }

    trait :with_attachment do
      attachment { File.new("#{Rails.root}/spec/files/attachment.jpg") }
    end
  end
end
```

Now, if we use `FactoryBot.create(:note, :with_attachment)` in a test, we'll get a new note object with a file attached.

Use this judiciously! Each use of this factory will perform a write to the file system, meaning a slower test.

Testing background workers

The imaginary marketing department for the imaginary company behind our project management application has asked that we collect location information on users, to help target broadcast and print advertising. Leaving it to the legal team to sort out the privacy and terms of service details, we've implemented the feature by creating a `location` attribute on the User model, and populating it with a city, state, and country as provided by an external geocoding service when a user signs in. This extra step runs in the background, so we don't keep the user waiting for the process to complete, via Active Job.

Although this feature is implemented in our sample application, it's a little tricky to see in action in the Rails development environment, since sign-ins will likely come in via `localhost` or `127.0.0.1`. I've added seed data with fake, random IP addresses for you to experiment with in the Rails console.

First, run `bin/rails db:seed` to add the seed data to your development environment.

Then, open `bin/rails console`, select one of the users, and call the `geocode` method on it:

```
> u = User.find(10)
> u.location
 => nil
> u.geocode
 => true
> u.location
 => "Johannesburg, Gauteng, South Africa"
```

Performing slow processes like this in the background is a relatively simple and effective way to keep web applications performant, but moving those processes away from the main flow means we need to consider how to test them a little differently. Luckily, Rails and rspec-rails provide nice support for testing Active Job workers, at varying levels.

Let's start with an integration test. We don't have one explicitly for user sign-in yet, so create one with `bin/rails g rspec:feature sign_in`. Then fill in the details:

spec/features/sign_ins_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Sign in", type: :feature do
  let(:user) { FactoryBot.create(:user) }

  before do
    ActiveJob::Base.queue_adapter = :test
  end

  scenario "user signs in" do
    visit root_path
    click_link "Sign In"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"

    expect {
      GeocodeUserJob.perform_later(user)
    }.to have_enqueued_job.with(user)
  end
end
```

There are two chunks of code here directly relating to Active Job and our background geocoding process. First, rspec-rails requires that we specify the `:test` queue adapter in order to test background jobs. Without this, the test will raise a helpful exception:

```
StandardError:
  To use ActiveJob matchers set `ActiveJob::Base.queue_adapter = :test`
```

I've included it in a `before` block here to help it stand out, but it could also be included inline within the `scenario`, since this is a one-off test. You could also experiment with DRY testing techniques from chapter 8, should you have tests across multiple files requiring the test queue.

Next, we need to make sure that the job actually gets added to the queue. rspec-rails provides a few matchers for this–here, we'll use `have_enqueued_job` to check that the correct job has been called, with the correct input. Note that the matcher must be contained within a block-style `expect`–if it's not, the test raises another helpful exception:

```
ArgumentError:
  have_enqueued_job and enqueue_job only support block expectations
```

The `have_enqueued_job` matcher also allows chaining, or setting details like a queue priority or scheduled time. The matcher's online documentation[49] has a rich set of examples.

[49]http://www.rubydoc.info/gems/rspec-rails/RSpec%2FRails%2FMatchers%3Ahave_enqueued_job

Now that we’ve checked that the background job is correctly wired with the rest of the app, let’s test at a lower level, and ensure that the job is calling the correct code within the application. To do this, we’ll create a new test file specific to the job:

```
bin/rails g rspec:job geocode_user
```

The new file looks similar to others we’ve generated throughout this book:

spec/jobs/geocode_user_job_spec.rb

```ruby
require 'rails_helper'

RSpec.describe GeocodeUserJob, type: :job do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

Let’s first add a test to make sure the job calls the `geocode` method on the user. Replace the `pending` test with a new one:

spec/jobs/geocode_user_job_spec.rb

```ruby
require 'rails_helper'

RSpec.describe GeocodeUserJob, type: :job do
  it "calls geocode on the user" do
    user = instance_double("User")
    expect(user).to receive(:geocode)
    GeocodeUserJob.perform_now(user)
  end
end
```

In this test, we use an `instance_double`, as introduced in chapter 9, to create a mock user for the test. Then we let RSpec know that this mock user should receive a call to the `geocode` method at some point in the test. Finally, we call the background job itself, via `perform_now`–that way, it doesn’t get queued, so we can test for results immediately.

The geocode instance method is provided by the Geocoder gem. In our app, it's put to use in the User model, using the geocoded_by method. In-depth knowledge of Geocoder isn't necessary for the purposes of this test, but you may find its documentation enlightening[50] if you decide to practice testing by adding more coverage.

Testing email delivery

Most full stack web applications send some sort of email messages to users. In our case, we send a short welcome message, with a reminder of how to use the application in the future. We can test that at two levels–first, that the message is crafted properly; and second, that it's wired up in the correct places to actually deliver. We'll build on our new knowledge of testing background workers here, since mail delivery is commonly handled this way.

Let's start with a test focusing just on the mailer, which is located in *app/mailers/user_mailer.rb*. At this level, we should test that the sender and recipient addresses are correct, and that the subject and message body contain anything important. (In more complex mailers, test coverage will likely check for more things than our basic example.)

We can use a generator to create a new test file:

```
bin/rails g rspec:mailer user_mailer
```

The new file of interest is added at *spec/mailers/user_mailer.rb* and contains the following boilerplate:

[50]http://www.rubygeocoder.com

spec/mailers/user_mailer_spec.rb

```ruby
require "rails_helper"

RSpec.describe UserMailer, type: :mailer do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

Let's fill it in:

spec/mailers/user_mailer_spec.rb

```ruby
require "rails_helper"

RSpec.describe UserMailer, type: :mailer do
  describe "welcome_email" do
    let(:user) { FactoryBot.create(:user) }
    let(:mail) { UserMailer.welcome_email(user) }

    it "sends a welcome email to the user's email address" do
      expect(mail.to).to eq [user.email]
    end

    it "sends from the support email address" do
      expect(mail.from).to eq ["support@example.com"]
    end

    it "sends with the correct subject" do
      expect(mail.subject).to eq "Welcome to Projects!"
    end

    it "greets the user by first name" do
      expect(mail.body).to match(/Hello #{user.first_name},/)
    end

    it "reminds the user of the registered email address" do
      expect(mail.body).to match user.email
    end
  end
end
```

We start by setting up some test data–a new user, and the mailer to test. From there, we add unit tests for key bits of functionality, starting with the `mail.to` address. Note that this value is an array of strings, not a single string. This applies to the `mail.from` value as well. We could've used RSpec's `contain` matcher here, but I prefer to check for array equality, and avoid the possibility of extra recipients or senders being added.

The test for `mail.subject` is hopefully straightforward; we've used the `eq` matcher to compare strings plenty of times by now. But the final two tests, covering `mail.body`, are a little different in that they use the `match` matcher–in these cases, we don't need to test the entire message body, just a couple of key pieces. In the first example, we've got a regular expression in place to ensure that the message includes a friendly greeting (like *Hello, Maggie,*). The second example also uses `match`, but only ensures that the string `user.email` is found somewhere within the body–no regular expression required, in this case.

If you want more expressiveness in your mail-related specs, check out the Email Spec[51] library. It adds matchers like `deliver_to` and `have_body_text`. You may find that it makes your tests more readable.

Again, the complexity of your own mailer specs will depend on the complexity of your mailers. In this case, we've probably covered what we need, but we could certainly add more if doing so increases our confidence in the mailer.

In the meantime, though, let's turn our attention to testing the mailer within the larger context of the application. Whenever a new user gets created, the welcome email message should be delivered. How can we validate that actually happens? We could test at a high level with an integration test, or attempt to test a little lower at a model level. As an exercise, we'll look at both options, starting with the integration test.

The message is delivered as part of the user sign-up workflow, so let's create a new feature spec for that:

```
bin/rails g spec:feature sign_up
```

Next, let's add the steps a user will take to sign up, and the expected outcomes, to the new file:

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

spec/features/sign_ups_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Sign-ups", type: :feature do
  include ActiveJob::TestHelper

  scenario "user successfully signs up" do
    visit root_path
    click_link "Sign up"

    perform_enqueued_jobs do
      expect {
        fill_in "First name", with: "First"
        fill_in "Last name", with: "Last"
        fill_in "Email", with: "test@example.com"
        fill_in "Password", with: "test123"
        fill_in "Password confirmation", with: "test123"
        click_button "Sign up"
      }.to change(User, :count).by(1)

      expect(page).to \
        have_content "Welcome! You have signed up successfully."
      expect(current_path).to eq root_path
      expect(page).to have_content "First Last"
    end

    mail = ActionMailer::Base.deliveries.last

    aggregate_failures do
      expect(mail.to).to eq ["test@example.com"]
      expect(mail.from).to eq ["support@example.com"]
      expect(mail.subject).to eq "Welcome to Projects!"
      expect(mail.body).to match "Hello First,"
      expect(mail.body).to match "test@example.com"
    end
  end
end
```

About two-thirds of this test may look familiar after chapter 6–we are using

Capybara to simulate completing the sign-up form. The remaining third focuses on the email delivery. Since mail is delivered as a background process, we need to wrap the test in a `perform_enqueued_jobs` block. This helper is provided by the `ActiveJob::TestHelper` module, which we include at the beginning of the spec file.

With that in place, we can now access `ActionMailer::Base.deliveries` and grab its last value. In this case, it's the welcome message delivered after the user filled in the registration form. Once we've got a `mail` to test, the remaining expectations are almost identical to the unit tests we just added on the mailer.

In fact, this level of detail may not be necessary at an integration level–verifying that the message delivers to the correct user (by checking `mail.to`) and is the correct message (by checking `mail.subject`) may suffice here, keeping the remaining details in the mailer spec. I just wanted to demonstrate that RSpec gives us options.

We could also test the integration directly at the point of contact between the User model and UserMailer mailer. As the application is written, this happens in an `after_create` callback when a new User is initially added, so we could add coverage in our existing User model spec:

spec/models/user_spec.rb

```ruby
require 'rails_helper'

RSpec.describe User, type: :model do
  # Earlier examples omitted ...

  it "sends a welcome email on account creation" do
    allow(UserMailer).to \
      receive_message_chain(:welcome_email, :deliver_later)
    user = FactoryBot.create(:user)
    expect(UserMailer).to have_received(:welcome_email).with(user)
  end
end
```

In this test, we're first using `receive_message_chain` to stub the `deliver_later` method that Action Job gives us on `UserMailer.welcome_email`, similar to how we used this method in chapter 9.

Next, we need to create a user to test. Since the mailer is triggered as part of the user creation process, we can test it straightaway with a **spy**. Spies are similar to the test doubles we explored in chapter 9, with the exception that they confirm something happened *after* the code to test was run, by asking if the class (`UserMailer`) received the message (`:welcome_email`) with an expected object (`user`), via `have_received`.

Why is this necessary? It's to work around the fact that we need to create `user` and assign a variable to it, but doing so also triggers the mailer we're interested in testing.

In other words, this won't work:

```
it "sends a welcome email on account creation" do
  expect(UserMailer).to receive(:welcome_email).with(user)
  user = FactoryBot.create(:user)
end
```

It fails with

```
Failures:

  1) User sends a welcome email on account creation
     Failure/Error: expect(UserMailer).to
       receive(:welcome_email).with(user)

     NameError:
       undefined local variable or method `user' for
       #<RSpec::ExampleGroups::User:0x007fca733cb578>
```

We can't use the value of `user` before it's created, but we can't create `user` without triggering the code to test. Luckily, spies give us an alternative workflow.

I like that this test does *just* enough to make sure the mailer is called at the right place, with the right information. It doesn't require the formality of creating a user through the web interface, so it's much faster. On the downside, it relies on a method provided by RSpec for testing legacy code[52], and the use of a spy may catch beginning developers by surprise at first.

You may also consider taking the opportunity to restructure the workflow, so that rather than using `deliver_later` to send the welcome message, email delivery is outsourced to a separate background job, or remove the `after_create` callback. In

[52]https://relishapp.com/rspec/rspec-mocks/docs/working-with-legacy-code/message-chains

such cases, I advise keeping the high-level integration test in place, then maybe getting rid of it once you find that lower-level tests sufficiently verify that the application code is wired together correctly. Again, as developers, we have options.

Testing web services

Let's get back to geocoding now. With coverage for the background job in place, we're still missing a detail of the implementation–is the geocoding process *actually* happening? As it's currently coded, geocoding takes place in the background, after a user has successfully signed in, but only if they're not logging in from the same host that's running the application server. We request location information based on an IP address. And again, geocoding isn't performed within our actual application–rather, we outsource it to an external geocoding service, via an HTTP call.

Let's add a test that reaches out to the geocoding service. Inside the existing User model spec, add a new test:

spec/models/user_spec.rb

```ruby
require 'rails_helper'

RSpec.describe User, type: :model do
  # earlier tests omitted ...

  it "performs geocoding" do
    user = FactoryBot.create(:user, last_sign_in_ip: "161.185.207.20")
    expect {
      user.geocode
    }.to change(user, :location).
      from(nil).
      to("Brooklyn, New York, US")
  end
end
```

This test looks similar to others we wrote way back in chapter 3: We create a user with a pre-assigned static address, geocode the user, then verify that the user's location value has been established. (I happen to know the location for this IP address, based on earlier experimentation with geocoding.)

Run the spec, and it passes–but there's a problem. This one little spec takes significantly longer to run than the rest of the file combined. Can you guess why? Since we're making a live HTTP request to the geocoding service, we have to wait for the service to return a location before populating the new value.

In addition to potentially significant slowdowns, testing directly against external services can have other costs. If the external API is rate-limited, you may see tests intermittently begin to fail once the rate limit has been passed. Or if the service charges for access, you may see an actual financial cost associated with the test!

The VCR[53] gem is a great tool for mitigating these issues, keeping your tests speedy and your API request calls at a minimum. VCR works by watching for external HTTP requests coming from your Ruby code. When it comes across a test that requires such a request, it causes the test to fail instantly. In order to make it pass, you'll need to create a *cassette* onto which to record the HTTP transaction. Run the test again, and VCR captures the request and response into a file. Now, future tests making the same request will use data from the file, instead of making another network request to the external API.

Let's start by adding VCR to the app. First, include the new gem and its dependency, WebMock, to the *Gemfile*:

Gemfile

```
group :test do
  # other testing gems omitted ...
  gem 'vcr'
  gem 'webmock'
end
```

WebMock[54] is the HTTP stubbing library that VCR will use behind the scenes when it records each transaction. WebMock is a powerful tool in its own right, but in order to keep things simple, we won't get into its details here. Run `bundle install` to add the new dependencies.

Next, we need to configure RSpec to use VCR in tests involving outgoing HTTP requests. Create a new file, *spec/support/vcr.rb*, and fill in its contents:

[53]https://github.com/vcr/vcr
[54]https://github.com/bblimke/webmock

spec/support/vcr.rb

```ruby
require "vcr"

VCR.configure do |config|
  config.cassette_library_dir = "#{::Rails.root}/spec/cassettes"
  config.hook_into :webmock
  config.ignore_localhost = true
  config.configure_rspec_metadata!
end
```

In this new configuration file, we're specifying that *cassettes*, or recordings, will be stored in the *spec/cassettes* directory of our app. As already mentioned, we'll use WebMock to perform the stubbing (though VCR offers support for a number of other stubbing libraries). We'll ignore requests to `localhost`, such as the AJAX call to check off tasks as complete. And finally, we'll allow VCR to be activated by an RSpec tag–similarly to how we set some tests to require a JavaScript-capable browser with `js: true`, we can enable VCR per test with `vcr: true`.

With configuration complete, run the new test again and see what happens. The test runs much more quickly, but now we get a failure:

```
Failures:

  1) User performs geocoding
     Failure/Error: user.geocode

     VCR::Errors::UnhandledHTTPRequestError:


       ================================================================
       An HTTP request has been made that VCR does not know how to handle:
         GET https://freegeoip.io/json/161.185.207.20

       There is currently no cassette in use. There are a few ways
       you can configure VCR to handle this request:
```

With VCR in place, tests that make external HTTP requests will fail with an `UnhandledHTTPRequestError` exception. To rectify, add that `vcr: true` option to the test:

spec/models/user_spec.rb

```
it "performs geocoding", vcr: true do
  user = FactoryBot.create(:user, last_sign_in_ip: "161.185.207.20")
  expect {
    user.geocode
  }.to change(user, :location).
    from(nil).
    to("Brooklyn, New York, United States")
end
```

Re-run the test, and two things should happen: First, it passes now. And second, it's recorded the request and response to a file in the *spec/cassettes* directory, as we configured a moment ago. Open the file to take a look. It's a YAML file containing the parameters of the request, followed by the parameters returned back from the geocoding service. Run the test again–it will still pass, but this time, it uses the contents of the cassette file as stubs for a real HTTP request and response.

In this example, we've used VCR to record a transaction performed in a model spec, but it can be used in any test, at any layer, that performs an HTTP transaction as part of the test.

I like VCR a lot, but it has its downsides. In particular, watch for cassettes to become *stale*–that is, if the external API you're testing against changes, you have no way of knowing when a stale cassette is in place. The only way to know for sure is to delete the cassette and re-run the test. Many Rails developers opt to omit cassette files from their projects' version control, so new developers *have* to record their cassettes on the first test suite run. If you decide to do this, you may also opt to automatically re-record cassettes on a regular interval[55], to help spot breaking API changes sooner.

A second concern is keeping sensitive information like secret API tokens and users' personal information out of the cassettes. VCR provides sanitization options[56] for scrubbing these values out of recordings before they're saved to file. This is especially important if you *do* opt to keep cassettes in version control.

[55]https://relishapp.com/vcr/vcr/v/3-0-3/docs/cassettes/automatic-re-recording
[56]https://relishapp.com/vcr/vcr/v/3-0-3/docs/configuration/filter-sensitive-data

Summary

Even though things like email, file uploads, web services, and background processors 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. If your own app doesn't have such functionality, try testing the Devise password reset integration in the sample application.
- Does your application have any file upload functionality or background jobs? 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?

11. Toward test-driven development

Whew. We've come a long way with our project management 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 approach–tests come first, then the code to make those tests pass, then refactoring to make our code more resilient for the long haul. Along the way, we use tests to inform coding choices, in an effort to produce bug-free software that may be updates in the future, as new requirements arise.

Let's try TDD now in our sample application!

You can view all the code changes for this chapter in a single diff[57] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-11-tdd origin/10-testing-the-rest
```

Defining a feature

We currently provide no way to mark projects as complete. It would be nice to address this, so that eventually our users can file away finished work, and only focus on current projects. Let's address this now, by adding a couple of new features:

[57] https://github.com/everydayrails/everydayrails-rspec-2017/compare/10-testing-the-rest...11-tdd

- Add a button to set a project as *complete*.
- Don't show completed projects on the dashboard we show users upon sign-in.

Let's start with the first scenario. Before coding, start by running the full test suite, and make sure everything's green 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. We've already got a feature file for managing projects, so we can add a new scenario to it. (In some cases, you may find it makes more sense to create a new file.) Open the existing file, and stub in the new scenario:

spec/features/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.feature "Projects", type: :feature do
  scenario "user creates a new project" do
    # original scenario omitted ...
  end

  scenario "user completes a project"
end
```

Save the file, and run the specs in it using `bin/rspec spec/features/projects_-spec.rb`. As you should hopefully expect by now, we get the following feedback from RSpec:

```
Projects
  user creates a new project
  user completes a project (PENDING: Not yet implemented)

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

  1) Projects user completes a project
     # Not yet implemented
     # ./spec/features/projects_spec.rb:26


Finished in 0.7601 seconds (files took 0.35072 seconds to load)
2 examples, 0 failures, 1 pending
```

Let's add some steps to the new scenario, describing the process a user will follow to complete a project. To start, let's think through what we'll need, what the user will do, and what the expected outcome of those actions will be:

1. We need a user with a project, and that user must be signed in.
2. The user will visit the project page and click a *complete* button.
3. The project will be marked as *completed*.

I sometimes begin a new test by including that information as comments within the test, to be quickly replaced with test code:

spec/features/projects_spec.rb

```ruby
scenario "user completes a project" do
  # given a user with a project
  # and that the user is logged in
  # when the user visits the project page
  # and the user clicks the "complete" button
  # then the project is marked as complete
end
```

Let's write some test code now, replacing the comments with what we've learned by this point:

spec/features/projects_spec.rb

```ruby
scenario "user completes a project", focus: true do
  user = FactoryBot.create(:user)
  project = FactoryBot.create(:project, owner: user)
  sign_in user

  visit project_path(project)
  click_button "Complete"

  expect(project.reload.completed?).to be true
  expect(page).to \
    have_content "Congratulations, this project is complete!"
  expect(page).to have_content "Completed"
  expect(page).to_not have_button "Complete"
end
```

I've included the `focus: true` tag here so that `bin/rspec` only runs that spec, and others I may add while working on a new feature, instead of running the entire suite each time. If you do the same, remember to remove the tag and run the full suite prior to committing any changes. See chapter 9 from more about using tags to speed up test runs. You can also use `:focus` as shorthand for `focus: true`.

Even though we haven't written the actual application code to make this new feature a reality, we've already indicated a few things about how it might work. First, there will be a button labeled *Complete*. Clicking that button will update a boolean attribute named `completed` on the project, changing it from `false` to `true`. We'll notify the user with a flash message that the process is complete, and make sure the button is removed, in favor of a label indicating that this project is finished. This is the essence of *test-driven development*. By way of writing a test first, we're actively thinking about how the code it's testing should behave.

From red to green

Run the specs again. 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) Projects user completes a project
   Failure/Error: click_button "Complete"

   Capybara::ElementNotFound:
     Unable to find button "Complete"

     1) Projects user completes a project
        Failure/Error: click_button "Complete"

        Capybara::ElementNotFound:
          Unable to find button "Complete"
```

Now, what's the simplest thing we can do to move forward? What if we try just adding a new button to the view?

app/views/projects/show.html.erb

```
<h1 class="heading">
  <%= @project.name %>
  <%= link_to edit_project_path(@project),
      class: "btn btn-default btn-sm btn-inline" do %>
   <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
   Edit
  <% end %>
  <button class="btn btn-default btn-sm btn-inline">
    <span class="glyphicon glyphicon-ok" aria-hidden="true"></span>
    Complete
  </button>
</h1>
<!-- rest of view omitted ... -->
```

Let's run the test again, and check our progress:

```
1) Projects user completes a project
   Failure/Error: expect(project.reload.completed?).to be true

   NoMethodError:
     undefined method `completed?' for #<Project:0x007fbcc80c94f0>
```

RSpec is giving us a hint about what to do next: Add a method named `completed?` to the Project model. Now, depending on the application and its business logic, this could mean a few things. Is a project *complete* when all of its tasks are done? If that's the case, then we could move forward by defining a `complete?` method on the Project model, then returning whether a given project's collection of incomplete tasks is empty (so, the project is complete) or not (meaning the project is incomplete).

But in our case, we know that *completeness* will be determined explicitly by a user action–clicking a button–and that this completion status should be persisted. So it looks like we'll need to add a new Active Record attribute to the model, to store this value. Create a migration to add the column to the `projects` table, then run the migration:

```
bin/rails g migration add_completed_to_projects completed:boolean
bin/rails db:migrate
```

Rails tries to automatically apply new migrations to your application's test database, but it doesn't always succeed. If you get an error message telling you to run the migration on the test database first, follow the provided instructions (`bin/rails db:migrate RAILS_ENV=test`), then re-run your test.

Run the new spec again with the change in place. Another failure, but it's a new one–that means we're making progress.

```
1) Projects user completes a project
   Failure/Error: expect(project.reload.completed?).to be true

     expected true
          got false
```

Unfortunately, this high-level test as informative as other have been. You may already have figured out what's going on, but let's check in with Launchy, which we covered in chapter 6. Temporarily insert launchy in the test:

spec/features/projects_spec.rb

```
scenario "user completes a project", focus: true do
  user = FactoryBot.create(:user)
  project = FactoryBot.create(:project, owner: user)
  login_as user, scope: :user

  visit project_path(project)
  click_button "Complete"
  save_and_open_page
  expect(project.reload.completed?).to be true
  expect(page).to \
    have_content "Congratulations, this project is complete!"
  expect(page).to have_content "Completed"
  expect(page).to_not have_button "Complete"
end
```

That's interesting. We're still on the project page, like clicking the button didn't do anything. Oh yeah, we put a basic `<button>` tag into the view a few moments ago, but we need to make it actually work. Let's do that now by updating the view.

We've got another design decision to make, though–how best to route the button, so that clicking it updates the database? We could keep things simple and re-use the `update` action on the projects controller, but since we need slightly different behavior–in this case, a different flash message–let's try an additional member action on the controller. Once we've got passing test coverage of the new feature, we can try as many alternate implementations as time allows.

At this point, I like to start at a high level, and work down. Let's go back to the view, and fix the button. Replace the `<button>` tag with a call to the Rails `button_to` helper:

app/views/projects/show.html.erb

```
<h1 class="heading">
  <%= @project.name %>
  <%= link_to edit_project_path(@project),
      class: "btn btn-default btn-sm btn-inline" do %>
    <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
    Edit
  <% end %>
  <%= button_to complete_project_path(@project),
      method: :patch,
      form: { style: "display:inline-block;" },
      class: "btn btn-default btn-sm btn-inline" do %>
    <span class="glyphicon glyphicon-ok" aria-hidden="true"></span>
    Complete
  <% end %>
</h1>
<!-- rest of view omitted ... -->
```

In addition to a little extra style, we've also defined what we think the route helper for this new action should be called: `complete_project_path`. Remove `save_and_open_page` from the spec, and run it again. Ah, a new failure:

```
1) Projects user completes a project
   Failure/Error: <%= button_to complete_project_path(@project),

   ActionView::Template::Error:
     undefined method `complete_project_path' for
     #<#<Class:0x007fc40faadfb8>:0x007fc40faac230>
     Did you mean?  compute_asset_path
```

Okay, we *didn't* mean `compute_asset_path`, but this *is* a hint that we haven't defined the new route we want to use. So let's add the route to the routes file now:

config/routes.rb

```ruby
resources :projects do
  resources :notes
  resources :tasks do
    member do
      post :toggle
    end
  end
  member do
    patch :complete
  end
end
```

Running the spec after this addition gives us another new failure:

```
1) Projects user completes a project
   Failure/Error: click_button "Complete"

   AbstractController::ActionNotFound:
     The action 'complete' could not be found for ProjectsController
```

RSpec's given us another good hint about what to fix. Let's go ahead and do as it suggests, and create a skeletal `complete` action on that controller, beneath the existing `destroy` action that the Rails generator provided:

app/controllers/projects_controller.rb

```ruby
def destroy
  # Rails generator code omitted ...
end

def complete
end
```

It's tempting to start filling it in, but a tenet of test-driven development is *write as little code as necessary to move the test forward.* The test complained about a missing action, and we've added it. Let's see where things stand now with another test run:

```
1) Projects user completes a project
   Failure/Error: unless @project.owner == current_user

   NoMethodError:
     undefined method `owner' for nil:NilClass
   # ./app/controllers/application_controller.rb:13:in `project_owner?'
```

That's interesting–we're now failing in a totally different controller! Why is that? We've got a couple of callbacks that run before some actions on the projects controller, but we haven't included the new `complete` action. Let's add that now.

app/controllers/projects_controller.rb

```ruby
class ProjectsController < ApplicationController
  before_action :set_project,
    only: [:show, :edit, :update, :destroy, :complete]
  before_action :project_owner?, except: [:index, :new, :create]

  # rest of controller omitted ...
```

And run the specs again:

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(project.reload.completed?).to be true
```

It would seem we've regressed a few steps–didn't we see this failure a minute ago?, But in reality, we're getting close. The failure before this told us that the code gets to the controller action, but nothing happens after that. Time to fill in the details in that `complete` action. Let's add enough code to satisfy the happy path, in which we don't have any problems blocking the user from completing a project:

app/controllers/projects_controller.rb

```ruby
def complete
  @project.update_attributes!(completed: true)
  redirect_to @project,
    notice: "Congratulations, this project is complete!"
end
```

Run the spec again–looks like we're getting close to wrapping this up!

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(page).to have_content "Completed"
       expected to find text "Completed" in "Toggle navigation
       Project Manager Projects Aaron Sumner Sign Out
       Ã– Congratulations, this project is complete!
       Project 1 Edit Complete A test project. Owner: Aaron Sumner
       Due: October 11, 2017 (7 days from now) Tasks Add Task
       Name Actions Notes Add Note Term"
```

We can read through the contents of the page in this case, to see that the word *Completed* indeed doesn't appear there. That's because we haven't added it–we're performing test-driven development, after all. Let's make an addition to the view, adding a `<span>` with a prominent label next to the button we created earlier:

app/views/projects/show.html.erb

```erb
<h1 class="heading">
  <%= @project.name %>
  <%= link_to edit_project_path(@project),
      class: "btn btn-default btn-sm btn-inline" do %>
    <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
    Edit
  <% end %>
  <%= button_to complete_project_path(@project),
      method: :patch,
      form: { style: "display:inline-block;" },
      class: "btn btn-default btn-sm btn-inline" do %>
    <span class="glyphicon glyphicon-ok" aria-hidden="true"></span>
    Complete
  <% end %>
  <span class="label label-success">Completed</span>
</h1>
<!-- rest of view omitted ... -->
```

Run the test again. Only the last step is failing!

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(page).to_not have_button "Complete"
       expected not to find button "Complete", found 1 match: "Complete"
```

Even though the project is complete, we're still showing the *Complete* button. Since this could confuse users, we should not display the button in the interface when viewing a completed project. We can address that with another change in the view:

app/views/projects/show.html.erb

```
<h1 class="heading">
  <%= @project.name %>
  <%= link_to edit_project_path(@project),
      class: "btn btn-default btn-sm btn-inline" do %>
    <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
    Edit
  <% end %>
  <% unless @project.completed? %>
    <%= button_to complete_project_path(@project),
        method: :patch,
        form: { style: "display:inline-block;" },
        class: "btn btn-default btn-sm btn-inline" do %>
      <span class="glyphicon glyphicon-ok" aria-hidden="true"></span>
      Complete
    <% end %>
  <% end %>
  <span class="label label-success">Completed</span>
</h1>
```

Run the spec–and we're passing!

```
Projects
  user completes a project

Finished in 0.58043 seconds (files took 0.35844 seconds to load)
1 example, 0 failures
```

At this point, especially when my code changes involve working changes to logic in a Rails view, I like to check my work in the browser. (Or if I'm writing an API endpoint, test it with curl or another HTTP client.) Start a Rails development server on your computer, if it's not already running, and open a project page in your browser. Clicking the *Complete* button does indeed mark the project as complete, and hides the button–but that *Completed* label we added is there, regardless of the project's completed status! Looks like we need to tweak the test a little bit before calling this new feature finished. Let's check the view to make sure *Completed* does *not* appear on the page prior to performing the new action:

spec/features/projects_spec.rb

```ruby
scenario "user completes a project" do
  user = FactoryBot.create(:user)
  project = FactoryBot.create(:project, owner: user)
  login_as user, scope: :user

  visit project_path(project)

  expect(page).to_not have_content "Completed"

  click_button "Complete"

  expect(project.reload.completed?).to be true
  expect(page).to \
    have_content "Congratulations, this project is complete!"
  expect(page).to have_content "Completed"
  expect(page).to_not have_button "Complete"
end
```

After all that work, we're back to a failing spec–but it's only temporary:

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(page).to_not have_content "Completed"
       expected not to find text "Completed" in "Toggle navigation
       Project Manager Projects Aaron Sumner Sign Out Project 1 Edit
       Complete Completed A test project. Owner: Aaron Sumner
       Due: October 11, 2017 (7 days from now) Tasks Add Task
       Name Actions Notes Add Note Term"
```

We can adapt the conditional we added around the button to also handle the label:

app/views/projects/show.html.erb

```
<h1 class="heading">
  <%= @project.name %>
  <%= link_to edit_project_path(@project),
      class: "btn btn-default btn-sm btn-inline" do %>
    <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
    Edit
  <% end %>
  <% if @project.completed? %>
    <span class="label label-success">Completed</span>
  <% else %>
    <%= button_to complete_project_path(@project),
        method: :patch,
        form: { style: "display:inline-block;" },
        class: "btn btn-default btn-sm btn-inline" do %>
      <span class="glyphicon glyphicon-ok" aria-hidden="true"></span>
      Complete
    <% end %>
  <% end %>
</h1>
```

And re-run the spec one more time:

```
Projects
  user completes a project

Finished in 0.82131 seconds (files took 0.47196 seconds to load)
1 example, 0 failures
```

Back to green!

We've been running a single spec, but it's always good to check the entire suite before going any further. Run `bin/rails rspec` and confirm that our changes haven't broken existing functionality. If you used `focus: true` or `:focus` tags during TDD, remove them now.

The suite is green; we're looking good!

Going outside-in

We drove out this new feature using a high-level integration test, making decisions about how the software should behave along the way. For the most part, the next bit of code to write was immediately clear, based on the feedback provided by RSpec–but a few times, we had to dig a little deeper to get clues about what was happening. In this case, Launchy worked reasonably well as a debugging tool, but often, you'll need to write more tests to understand the problem. These additional tests will poke at your code from different levels, and provide insights that aren't always possible to glean from higher-level tests. This is what *outside-in testing* is all about, and how I perform test-driven development.

I also like to take advantage of lower-level testing capabilities when the overhead of a simulated browser is overkill. The integration test we wrote covers the happy path, in which a user encounters no errors while completing a project. Is it necessary to write another high-level test to work through a failure? Maybe not–since we can already see that the *Complete* button is wired up correctly, and that the web interface properly displays a flash message, we may be able to drop down to a controller test and focus on the unique aspects of a failure: Making sure that the correct flash message gets set, and that the project object doesn't change. This time, since we've got code in place, we'll write a few controller tests to ensure that errors in completing a project are handled gracefully, too. We can add these to our existing project controller specs:

spec/controllers/projects_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  # other describe blocks omitted ...

  describe "#complete" do
    context "as an authenticated user" do
      let!(:project) { FactoryBot.create(:project, completed: nil) }

      before do
        sign_in project.owner
      end

      describe "an unsuccessful completion" do
        before do
          allow_any_instance_of(Project).
            to receive(:update_attributes).
            with(completed: true).
            and_return(false)
        end

        it "redirects to the project page" do
          patch :complete, params: { id: project.id }
          expect(response).to redirect_to project_path(project)
        end

        it "sets the flash" do
          patch :complete, params: { id: project.id }
          expect(flash[:alert]).to eq "Unable to complete project."
        end

        it "doesn't mark the project as completed" do
          expect {
            patch :complete, params: { id: project.id }
          }.to_not change(project, :completed)
        end
      end
    end
  end
end
```

For these examples to do their job, we have to simulate a failure. We use `allow_any_instance_of` to accomplish this. `allow_any_instance_of` is a variation of the `allow` method we used in chapter 9. In this case, we're intercepting calls to the `update_attributes` method on *any* project object, preventing it from saving the project's completion status. `allow_any_instance_of` can be disruptive, so we need to take care to *only* apply it to the tests within the `describe "an unsuccessful completion"` block.

Run the tests to see where things stand:

```
Failures:

  1) ProjectsController#complete as an authenticated user an
  unsuccessful completion sets the flash
     Failure/Error: expect(flash[:alert]).to eq "Unable to complete
     project."

       expected: "Unable to complete project."
            got: nil

       (compared using ==)
     # ./spec/controllers/projects_controller_spec.rb:246:in
     `block (5 levels) in <top (required)>'
```

So it looks like the user will be redirected as we expect, and the project won't be properly marked as completed. But we don't provide any message indicating a problem. Looking back at the application code, perhaps we can add a conditional to address this:

app/controllers/projects_controller.rb

```ruby
def complete
  if @project.update_attributes(completed: true)
    redirect_to @project,
      notice: "Congratulations, this project is complete!"
  else
    redirect_to @project, alert: "Unable to complete project."
  end
end
```

With that change, the new tests all pass. We've covered both the happy success path, as well as the failure path.

To summarize, when doing outside-in testing, I like to start with a high-level test to show the software behaving as intended, with all prerequisites and user inputs provided correctly (the *happy path*). In this case, that integration test came in the form of a feature test with browser simulation, but if I were testing an API, then I'd use a request spec (see chapter 7). Then, drop to a lower level of testing to ferret out the details as directly as possible. In this case, I used controller specs, but in many cases it's possible to test directly on a model. You may even use this as an opportunity to explore placing business logic outside of traditional Rails models and controllers, and into standalone classes like service objects.

I've written about using tests to help refactor toward service objects and similar patterns[58] in the Everyday Rails blog; actually doing so is beyond the scope of this book.

Let the feedback from the test guide you. If your current test isn't giving you sufficient feedback, try dropping down a level.

The red-green-refactor cycle

We've made the new feature work, but we're not done–with passing test coverage in place, we can explore improvements to our work. We've arrived at the *refactor*

[58]https://everydayrails.com/2017/11/20/replace-rspec-controller-tests.html

stage of *red-green-refactor*. With the new test coverage in place, we can consider alternative implementations, or just tidy up the one we just wrote.

Refactoring can be quite complex, and an in-depth introduction to it is beyond the scope of this book. However, here are some options to consider, sorted from least to most complex:

- We introduced a new conditional to the *projects/show* view, to determine whether to show the *Completed* button, or a message indicating that the project is complete. Should we extract that new code to a view partial, to keep the main view relatively simple?
- We talked briefly about different controller structures to implement the new completion route. Is the implementation we chose the best way to go? (After seeing what was involved with testing it, I'm not so sure–stubbing out Active Record methods in a controller test can be a hint that the controller is doing too much.)
- Maybe we could extract the business logic out of the controller and into something that's simpler to test–but where? Moving it to the Project model could be as simple as creating a new method `Project#mark_completed`, but that habit can lead to oversized models when taken too far. Another alternative might be to extract it to a *service object* that is only responsible for completing projects. Then we could test that new object directly, without having to run through a controller.
- Or we could choose an entirely different controller structure. Instead of adding an action to the existing controller, what if we created a new, nested resourceful controller that exposes an `update` action, responsible for completing the parent project?

Take advantage of your tests–and what they told you as you were writing code–to understand alternatives. There's almost always more than one way to code something, and they'll all have their pros and cons. As you refactor, try to make small, incremental changes, and keep your test suite green. That's the key to the refactoring step: *Any changes you make should result in the tests still passing.* (or, if they fail, it's only temporary). You may also find yourself darting up and down, from feature spec down to model, controller, and/or standalone Ruby object. It depends on where it makes the most sense to keep the code in your application, and which level of testing provides appropriate feedback.

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, as features are developed, saves much more time in the long haul. You’ve now got the tools you need to do the same in your own projects!

Exercises

- We implemented the first aspect of the new project completion feature together, but we still need to omit completed projects from the user’s dashboard. Try test-driving the feature–start with a new integration test that includes a complete and incomplete project, then shows (or does not show) the appropriate projects on the project owner’s dashboard.
- Once that feature is completed, how can a user access her completed projects? Consider how this might work, then implement it using TDD.
- If you feel up to it, try experimenting with one or more of the refactoring options listed in the *red-green-refactor* section. How do they change the nature of the tests, especially those handling failure cases?

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 and basic instance methods are often 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 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 or create a temporary branch in Git 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 removed my temporary code, wrote my specs, and then systematically reapplied my work.

As a general rule, I try to retype my work as opposed to just commenting and 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. 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.

Try to write integration 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 or request specs, you'll *start* with feature or request 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, and 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 little longer 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–whether it's integration testing, 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 your colleagues. 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 7, 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 toy Rails applications to be great for this purpose–create a new app (blogging app and to-do lists are always popular), 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 a blog that sends an email digest version to subscribers. If you want to hone your skills at testing external APIs, many services provide free or cheap developer tiers, perfect for practice. Don't wait for a feature request to arise in a production project–make up your own!

Goodbye, for now

You've now got all the tools you need to do basic automated testing in your Rails projects, using RSpec, Factory Bot, 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: https://facebook.com/everydayrails
- GitHub: https://github.com/everydayrails/everydayrails-rspec-2017/issues

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

Thanks again for reading.

Appendix A. Migrating to system specs

In 2017, literally days after I released a major update to this book with coverage of Rails 5.1 and RSpec 3.6, the RSpec team released a new version, 3.7. Due to how I build up each chapter incrementally, and the fact that the `rspec-rails` gem is introduced to this book's sample code base early on, I've decided not to try to retrofit the sample code. Instead, I want to share my experience upgrading the sample application to use the latest version of RSpec as of this writing (2018), version 3.8–and most importantly, its support for Rails 5.1 system tests.

With system tests, the Rails framework finally offers end-to-end, browser-based tests like the feature-style approach we explored in chapter 6, out of the box. These tests are called *system tests*[59].

Beginning with RSpec 3.7, you can add system specs to your suite as long as you're testing a Rails 5.1 application or newer. RSpec 3.7 will work with older versions of Rails, but you'll need to stick with feature tests for integration testing. You'll also miss out on some of the other features provided by Rails 5.1's system testing support, though there are alternatives I'll share later in this appendix.

You can view all the code changes for this appendix in a single diff[60] on GitHub.

If you'd like to follow along, follow the instructions in chapter 1 to clone the repository, then start with the previous chapter's branch:

```
git checkout -b my-appendix-a-system-tests origin/11-tdd
```

[59]http://guides.rubyonrails.org/testing.html#system-testing
[60]https://github.com/everydayrails/everydayrails-rspec-2017/compare/11-tdd...appendix-a-system-tests

The upgrade

I have not updated my Rails version for this appendix, but these configurations should also apply as-is to Rails 5.2.

Before starting the RSpec upgrade, run the existing specs and make sure everything still passes as expected. This is especially important on projects that have gone untouched for awhile. With a passing test suite, we can upgrade the dependency in the project's *Gemfile*:

Gemfile

```
group :development, :test do
  gem 'rspec-rails', '~> 3.8.0'
  # other gems ...
end
```

Run `bundle update rspec-rails` to complete the installation. If we wanted to, we could stop now. Feature specs continue to work with RSpec 3.8, and will continue to do so for the foreseeable future. However, in the interest of keeping our test suite forward-facing, let's move existing feature specs to system specs.

First, let's modify the test driver configuration. We'll explicitly set the speedy `Rack::Test` driver for basic browser testing, and a JavaScript capable driver for more complex browser interactions (we'll use `selenium-webdriver` and headless Chrome, as set up in chapter 6). Replace the file's contents:

spec/support/capybara.rb

```
RSpec.configure do |config|
  config.before(:each, type: :system) do
    driven_by :rack_test
  end

  config.before(:each, type: :system, js: true) do
    driven_by :selenium_chrome_headless
  end
end
```

You can also set `driven_by` on a test-by-test basis, but I like setting things system-wide when I can.

Let's wrap up by migrating existing feature specs to the new system testing structure and syntax. To begin, rename *spec/features* to *spec/system*, using the command line or your operating system's graphical interface.

Next, inside the new *spec/system* directory, change each spec's type from `type: :feature` to `type: :system`. For example:

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.feature "Projects", type: :system do
  # contents of spec file ...
end
```

In addition, we'll use `RSpec.describe` to define system specs, doing away with `RSpec.feature` on these same lines:

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  # contents of spec file ...
end
```

Optionally, replace the `scenario` alias in previous feature specs with the standard `it` syntax. This is how RSpec's documentation on Rails system specs is written. For example:

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  it "user creates a new project" do
    # example ...
```

I personally think this does not read like well-structured English, and would consider also changing the description to something like `it "creates a new project as a user"`. As an alternative, we could use nested `describe` or `context` blocks to indicate the test is following a workflow `as a user`. Or, we could just leave `scenario` in place for better readability. For the sake of keeping this transition simple, I'm going to stick with `scenario` for this example.

Finally, we need to configure Devise helpers to work in system specs instead of feature specs. Update the configuration for `Devise::Test::IntegrationHelpers`:

spec/rails_helper.rb

```
# Use Devise helpers in tests
config.include Devise::Test::ControllerHelpers, type: :controller
config.include RequestSpecHelper, type: :request
config.include Devise::Test::IntegrationHelpers, type: :system
```

Run the test suite, and these new system specs should pass. Going forward, we can add new integration tests in the *spec/system* directory.

Generating new system specs

As of version 3.8.0, `rspec-rails` doesn't provide a generator to create system specs, though a pull request is in progress as I write this. In the meantime, you'll need to manually create new spec files in *spec/system*, using this general boilerplate:

```ruby
require 'rails_helper'

RSpec.system "Test name", type: :system do
  it "does something useful" do
    # your test goes here ...
  end
end
```

Screenshots

Since we've been using Capybara, we can already save screenshots of browser-based tests in progress using `save_screenshot`, just as we've been using `save_and_open_page` since chapter 6. With Rails 5.1 and RSpec 3.7 or newer, we can use the alternative `take_screenshot` method anywhere in a test to generate an image of the simulated browser at that point. The only catch is, the test must be run with a JavaScript driver (in our case, that's `selenium-webdriver`). The file is saved in *tmp/screenshots* by default. And if a test fails, you'll get a screenshot automatically! This is a useful feature when debugging integration tests that run in headless environments.

If a test is using `Rack::Test` instead of a JavaScript driver, you can still use `save_page` to save an HTML file to *tmp/capybara*, or `save_and_open_page` to immediately open the file in a browser. See chapter 6.

If you're not yet using Rails 5.1, you can get similar functionality with the capybara-screenshot[61] gem. It requires a little additional setup, but is also more full-featured than what Rails 5.1 provides out of the box.

[61]https://github.com/mattheworiordan/capybara-screenshot

Database Cleaner

I removed coverage of Database Cleaner from the Rails 5.1 edition of the book, because it wasn't necessary to make tests work in the sample app. If you're upgrading an application from previous versions of Rails and RSpec, you may have used this tool to keep database state from leaking across tests. With Rails 5.1, you may no longer need Database Cleaner at all, at least from my limited experiments and what I've read about other developers' experiences.

More testing resources for Rails

While not exhaustive, the resources listed below can each play a role in giving you a better overall understanding of Rails application testing. I've reviewed them all myself.

RSpec

RSpec official documentation

We've focused on using RSpec with Rails in this book, but if you're interested in using it in non-Rails projects, the Relish docs are a great place to start. You'll find documentation on the latest RSpec releases, all the way back to version 2.13. https://relishapp.com/rspec

You'll also find documentation for rspec-rails on Relish. https://relishapp.com/rspec/rspec-rails/docs

Effective Testing with RSpec 3

The RSpec Book from Pragmatic Programmers had been out of date for awhile, but it's ben replaced by a totally new book written by current RSpec maintainer Myron Marston and Ian Dees. You'll find it useful if you need to use RSpec outside of Rails, or just want to reinforce your understanding of the framework. https://pragprog.com/book/rspec3/effective-testing-with-rspec-3

Better Specs

Better Specs is a really nice, opinionated collection of illustrated best practices to employ in your test suite. I don't agree with everything they suggest, but leave it to you to form your own opinions. http://betterspecs.org

RSpec the Right Way

Geoffrey Grosenbach of Pluralsight and Peepcode demonstrates the TDD process, using many of the same tools discussed in this book. Requires a Pluralsight subscription. https://www.pluralsight.com/courses/rspec-the-right-way

Railscasts

If you're new to Rails, you may not be familiar with Ryan Bates' top-notch screencast series, *Railscasts*. The series was highly influential to my own learning of Ruby on Rails, and is sorely missed today. Ryan did a number of episodes on testing; many either focus on RSpec or include it as part of a larger exercise. Some of the syntax is out-of-date, but the concepts still apply. Be sure to watch the episode "How I Test," which in part inspired this book. http://railscasts.com/?tag_id=7

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. https://groups.google.com/group/rspec

Rails testing

Rails 4 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. https://pragprog.com/book/nrtest2/rails-4-test-prescriptions

Noel has also done a number of talks on testing. Many are available on Confreaks at https://confreaks.tv/presenters/noel-rappin; slides at https://speakerdeck.com/noelrap.

Ruby Tapas

Ruby Tapas is my favorite subscription video service, with more than 500 episodes covering all aspects of Ruby. The testing-specific episodes are excellent, and some of them are free to view (though I *do* recommend investing in a subscription). https://www.rubytapas.com/category/testing/

Rails Tutorial

The book I wish had been around when I was learning Rails is Michael Hartl's *Rails Tutorial*. The third edition replaces RSpec with MiniTest, but don't let that dissuade you from checking it out–aside from setup and some syntax, the day-to-day differences between the two testing frameworks are not vastly different. I especially like Michael's pragmatic take on when to test first, and when to code first[62]. Also available as a series of screencasts, if that's your learning preference. https://ruby.railstutorial.org

Agile Web Development with Rails 5.1

Agile Web Development with Rails by Sam Ruby and David Bryant Copeland *is* the book that *was* available when I got started with Rails. https://pragprog.com/book/rails51/agile-web-development-with-rails-5-1

Learn Ruby on Rails

Daniel Kehoe's *Learn Ruby on Rails* focuses on getting started with programming in Rails, but includes a nice chapter on testing with MiniTest, an alternative to RSpec that's built into Rails. http://learn-rails.com

[62]https://www.railstutorial.org/book/static_pages#aside-when_to_test

About Everyday Rails

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

About the author

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

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

Colophon

The cover image of a practical, reliable, red pickup truck[63] is by iStockphoto contributor Habman_18[64]. 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.

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

Change log

April 7, 2019

Chapter 6:

- Replace usage of `chromedriver-helper` gem with `webdrivers`.

August 22, 2018

New:

- Add Appendix A, System specs in Rails 5.1

General:

- Drop "work in progress" messaging from README
- Update Devise to address a syntax error with Ruby 2.5 (no effect on the sample test code)
- Fix several code styling issues throughout book (thank you, Junichi Ito)

Chapter 6:

- Fix minor typo in ChromeDriver setup instructions

Chapter 8:

- Replace uses of `login_as` helper method provided by Warden, with Devise's `sign_in`

Chapter 10:

- Fix geocoding test example now that the geocoder gem now defaults to IPInfo.io for IP-based geocoding (a new version of geocoder is introduced in chapter 1's branch)
- Replace uses of `login_as` helper method provided by Warden, with Devise's `sign_in`

Chapter 11:

- Replace uses of `login_as` helper method provided by Warden, with Devise's `sign_in`

June 4, 2018

Chapter 1:

- Drop reference to old coverage of testing time

Chapter 2:

- Replace reference to Test::Unit with MiniTest
- Remove early reference to factories

Chapter 3:

- Fix broken line number settings in code samples
- Fix typo in user model spec generation instructions
- Add missing "$" prompt in project spec generation instructions
- Fix redundant use of `@note1`
- Fix pending spec count
- Reword description of `be_empty` matcher to work around Leanpub formatting issue

Chapter 4:

- Replace Factory Girl with Factory Bot
- Fix manuscript-only typo in code sample
- Improve language referring to User factory
- Fix reference to notes in callback section
- Replace `be_true` with `be_truthy` in hypothetical factory example

Chapter 6:

- Use chromedriver-helper to install chromedriver

Chapter 8:

- Prefer Devise feature helpers over Warden's
- Fix incorrect wording about expectations in aggregate failures description
- Fix small typo in support modules section
- Fix capitalization of `content_types`
- Fix typo in project setup context

Chapter 9:

- Fix indentation in integration testing dependencies

Chapter 10:

- Make instructions on requiring matchers less confusing
- Fix description of mail delivery integration tests
- Clean up explanation of `receive_message_chain`

Chapter 11:

- Fix code indentation in `complete` method
- Clarify language around testing outside of models and controllers
- Clarify language around `focus` vs. `focus: true`
- Clarify explanation of `completed?` method on project

- Fix small typo in section on the red-green-refactor cycle

More resources:

- Fix heading size for Ruby Tapas
- Use HTTPS for links to resources that have moved to it instead of HTTP
- Fix links to Relish documentation

November 27, 2017

- Spelling corrections only.

October 23, 2017

This is mostly an errata fix/maintenance release only. Thanks to Capybara maintainer Thomas Walpole for guidance on the Capybara-related issues:

- Update Capybara to take advantage of simplified configuration (Capybara now registers *selenium_chrome* and *selenium_chrome_headless* drivers without the extra config). If you've already worked through chapter 6, please run `bundle update capybara selenium-webdriver` to get the newer versions.
- Replace all instances of `Capybara.default_wait_time` with `Capybara.default_max_wait_time` (book only; this code is not used in the sample project itself)
- Update feature spec example to include at least one expectation inside an `expect {}`.
- Remove inaccurate bits about `find(...).click` in reference to waiting on JavaScript to complete.
- Fix warning `projects/app/views/projects/index.html.erb:4: warning: instance variable @project not initialized`
- Move test-specific gem dependencies to a standalone `:test` Gemfile group.
- Rename chapters 6 and 7 to make it more evident what layer of testing they cover.
- Mention ParallelTests as an option for speeding up test suites in chapter 9.
- Add Ruby Tapas to the resources appendix.

October 16, 2017

- Rewrite of chapter 11, *Toward Test-driven Development*, with commentary on outside-in testing and the red-green-refactor cycle.
- Fix broken apostrophes in chapter 8.

September 18, 2017

This release features a heavily rewritten version of chapter 10, *Testing the Rest*. If you worked through sample code in a RSpec 3.6/Rails 5.1 version of the book prior to this release, you'll need to add a new dependency on Faker to add some sample data to the project's development environment (though we're still not using Faker in tests). You'll also need to update your development and test database schemas to add support for some of these new tests. From your command line, run `bin/rails db:migrate`, then `RAILS_ENV=test bin/rails db:migrate` to update your schema.

- Rewrite section on testing file uploads.
- Add new section on testing background jobs.
- Rewrite section on testing email delivery.
- Rewrite section on testing external HTTP requests.

Other fixes:

- Replace usage of `Date.today` with `Date.current.in_time_zone` to work around issues with time comparisons, and follow general best practices of assuming time zones will get you into trouble.
- Fix incorrectly-named spec in chapter 3.

August 4, 2017

- Replace chapter *Speeding Up Tests* with *Writing Tests Faster, and Writing Faster Tests*, with coverage on RSpec's short syntax, Shoulda Matchers, keyboard shortcuts, test doubles, stubs, and tagging.
- Remove coverage of Guard since I don't use it anymore.
- Use RSpec's built-in mocking library instead of Factory Girl's `build_stubbed`. Nothing against the latter, it's just not as common in Rails code bases.

July 11, 2017

- Add a new chapter, *Keeping Tests DRY*, including coverage of support modules, `let`, shared contexts, custom matchers, aggregated failures, and readable tests.
- Drop coverage of shared examples. This used to be in a chapter on refactoring complicated controller specs, and was going to be in the new chapter. But I seldom use shared examples anymore, and controller testing is soft-deprecated.
- Add a new exercise to chapter 4.
- Fix incorrect path in a code sample in chapter 4.
- Fix incorrect factory setup in chapter 4.
- Remove extra data being passed in controller tests in chapter 5.
- Remove code formatting from samples that don't need it (for example, spec output and command line instructions).

June 18, 2017

This is the first iteration of a major rewrite for 2017. It addresses changes in RSpec and Rails, but more importantly, it addresses changes to my overall approach to testing. It uses a slightly more complex sample application, in order to better demonstrate techniques without running into merge conflicts during writing/development. I'm probably forgetting things, but here are some highlights of this release:

- Rewrite for RSpec 3.6 and Rails 5.1. Use versions of all gems current as of May and June, 2017.
- Replace sample application with a new project management app.
- Use Devise for authentication in sample app, instead of rolling our own.
- Remove coverage for Factory Girl's shorthand (`create` versus `FactoryGirl.create`), since I only use the long format now.
- Combine controller chapters into a single chapter, since controller testing is being phased out in Rails. Discussion of some of the DRY techniques previously covered here will be added to a new chapter on DRY specs.
- Use ChromeDriver to demonstrate JavaScript testing.
- Extract coverage of Database Cleaner from feature specs, for now, as I haven't need it yet in this sample app.
- Expand coverage of API testing via request specs, to its own chapter.

December 19, 2014

- Updated use of binstubs.
- Fix formatting for broken code sample in chapter 3.
- Prefer `not_to` in code samples (though it is interchangeable with `to_not`).
- Make sure code samples in the book refer to *spec/rails_helper.rb* as the primary configuration file.
- Fix phone factory example (chapter 4).
- Replace old references to PUT with the preferred PATCH.
- Rewrite DatabaseCleaner section (chapter 8).
- Add a little more information about tags (chapter 9).
- Skip specs with `skip`, not `pending`, and explain why (chapter 9).
- Fix formatting for mailer spec example (chapter 10).
- Fix file upload factory example (chapter 10).
- Use the `have_http_status` matcher when testing APIs (chapter 10).
- Use `is_expected_to` in terse syntax.
- Mention how to disable adding unwanted assets and helpers when using `rails scaffold` (chapter 11).
- Update additional resources listing.
- Minor changes throughout the book to improve readability.

October 2, 2014

- Release major update for RSpec 3.x and Rails 4.1 and beyond.
- Add content to chapter 10 to cover external service testing, API testing, and more.
- Drop the chapter on RSpec 2.99. If you need it, download the previous edition.
- Format changes for the PDF edition, to prepare for a possible print release at some point.
- Address various punctuation and grammatical issues that were bugging me.

April 25, 2014

- Add preview of chapter 12, upgrading to RSpec 2.99.

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 request specs (now feature specs).
- Other minor typo and language corrections throughout the book.
- Update `selenium-webdriver` version in chapter 8 to address incompatibility 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.