A practical approach to test-driven development

Testing with RSpec

everyday Rails

Aaron Sumner

Everyday Rails Testing with RSpec

A practical approach to test-driven development

Aaron Sumner

This book is available at https://leanpub.com/everydayrailsrspec

This version was published on 2025-09-15

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

Tweet This Book!

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

The suggested 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

Thank you for picking up this latest version of *Everyday Rails Testing with RSpec*. It's been awhile! As always, Rails and RSpec continue to evolve, but are still your best bet for quickly building (and testing) robust, scalable, full-stack web applications. But the biggest change to our field between 2017 (the last time I released a major update) and today is artificial intelligence.

Now, this is *not* a book about AI. In fact, this is the only time I'll even mention it. Can you use AI to write your tests? Sure. Can you expect them to be *good* tests? Sometimes. But how do you tell the good tests apart from the bad?

AI-assisted programming is likely not going away. And whether or not you choose to use new tools like Claude Code or GitHub Copilot in your daily workflow, we all need to keep core skills sharp: Clean, ideomatic code, fundamentals of software design, and test-driven development.

Good tests don't just prove your software works. Good tests also prove your software is sustainable for the long haul. Understanding how and where to write these tests takes time–and it's up to you, as the human in the room with your AI agentic programming buddy, to make those tests happen.

It's an exciting time to be a software developer! I hope *Everyday Rails Testing with RSpec* inspires you toward thoughtful software design and development.

Aaron Sumner
Saint Louis, MO, USA
September, 2025

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. That's pretty awesome, if you ask me.

Yet, many people developing in Rails are still 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. 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. As an engineer, 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 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. 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!

Confident testing

But early Rails books and tutorials focused more on development speed (build a blog in 15 minutes!) than on good practices like testing. If they covered testing at all, they generally reserved it for a chapter toward the end. Newer educational resources on Rails have addressed this shortcoming, and demonstrate how to test applications from the beginning. Countless books now exist *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. Such 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?

I have nothing against other Ruby testing frameworks. If the default MiniTest framework helps you confidently write sustainable test suites, that's great!

For me, RSpec's capacity for specs that are readable, without being overly 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 often 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. It 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 full-stack 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. My favorite is *Agile Web Development with Rails 7* by Sam Ruby; lots of people like Michael Hartl's *Rails Tutorial* as well. 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

If you're already familiar with using 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 maintenance and 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. 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; by you and your collaborators.

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 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," shipping to production, and hoping for the best. There's a better way: Taking advantage of a fully automated test suite and using tests to drive development and ferret out potential bugs and edge cases *before* users spot them.

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 7.1 and RSpec 3.12. Many of the concepts apply to older and newer versions of each, albeit with slightly different syntax.

The book starts with RSpec installation in an existing Rails application. From there, we build a test suite bit by bit, starting with small, isolated tests and working out to broader, more complex testing scenarios. This is the same, step-by-step process I used to get better at testing my own software. 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

You can grab the sample code at https://everydayrails.com/rspecbook. The provided Zip file contains snapshots of the sample application and its test suite as it grows, chapter to chapter. For example, the *02-models* directory includes the Rails application itself, and the tests added in chapter 2.

Code conventions

I'm using the following setup for this application:

- **Rails 7.1:** 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 6.0 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 3.2:** Any version of Ruby 3 should work.
- **RSpec 3.12:** RSpec 3.0 was released in spring, 2014, and has been largely stable since that time. Earlier versions used a significantly different syntax we won't cover in this edition of the book.

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 in the long run.

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/rspec-sample-rails-7.1/issues

A note about gem versions

The gem versions used in this book and the sample application are current as I write this Rails 7.1 edition, in 2024. 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.

A note about styling

Many of the code samples included in this book are based from code created by generators. As a result, you may see samples with mismatching styles–for example, use of single quotes and double quotes in the same file. In the interest of keeping differences between what's generated and what matters for learning RSpec, I've elected to leave styles in generated code as-is, but will otherwise generally follow conventions as defined by Standard Ruby.

About the sample application

Meet TasteDrivenDishes, the latest social site for finding and sharing your favorite recipes with users around the world! We'd better get a test suite in place before the site hits the front page of Hacker News–better for us to catch (and fix) bugs before our users and investors do.

To start, TasteDrivenDishes includes the following features:

- A user can browse recipes and filter by category.
- A user can create an account to add their own recipes.
- A signed-in user can mark recipes as favorites.
- A user's account has an avatar, provided by the Dicebear service.
- A developer can access a public API to develop external client applications.

Up to this point, I've intentionally avoided writing tests for the application (see the *01-untested* folder in the sample code download). This means I have a *test* directory full of untouched test files and data setup. I could run `bin/rails test`, 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!

2. Setting up RSpec

TasteDrivenDishes is currently *functioning*. At least we *think* it's functioning. Our only proof of that is we clicked through the links, made a few fake accounts and recipes, and added and edited data through the web browser.

Ship it, right?

Of course, this approach doesn't scale as we add features. Before we go any further, let's pause feature development and add an *automated test suite*, with RSpec at its core. Over the remainder of this book, we'll add coverage to TasteDrivenDishes, starting with RSpec, and adding other testing libraries as necessary to round out the suite.

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 start adding specs.

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

- We'll start by using Bundler to install RSpec.
- Next, we'll check for a test database and install one, if necessary.
- Finally, we'll configure RSpec to run the test suite!

To follow along with the code in this chapter, start with the *01-untested* folder from the sample code. Refer to the *02-setup* folder for the completed code from this chapter. See "Downloading the sample code" from Chapter 1 for download instructions.

Dependencies

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. If you don't already have a terminal command line open in the application, open one now. Then, at the command line prompt, type:

```
bundle add rspec-rails --version "~> 6.1.2" --group "development, test"
```

Note that we're only installing RSpec for use in the application's *development* and *test* environments. It won't be installed when deploying the application to production. We've also locked the version so that Bundler will install any version of the `rspec-rails` gem equal to or greater than 6.1.2, but not 6.2 or newer.

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 a 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.

Our application now has the first building block necessary to establish a solid test suite. Next up: Creating our test database.

Test database

For the purposes of teaching you about RSpec without much extra overhead, TasteDrivenDishes uses SQLite for its database backend.

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:

config/database.yml

```
1 test:
2   <<: *default
3   database: db/test.sqlite3
```

Or this if you're using MySQL or PostgreSQL:

config/database.yml

```
1 test:
2   <<: *default
3   database: recipes_test
```

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

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

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

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

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.

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.

The `rspec` binstub

Next, let's install a *binstub* for the RSpec test runner, just to save ourselves from a bit of typing. We'll be running the test suite a lot! On your command line, generate the binstub:

```
1 bundle binstubs rspec-core
```

This will create an *rspec* executable, inside the application's *bin* directory. If you don't want to install the binstub 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.

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:

```
$ bin/rspec
```

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

```
No examples found.

Finished in 0.00015 seconds (files took 0.0484 seconds to load)
0 examples, 0 failures
```

If your output looks different, go back and make sure you've followed the steps outlined above.

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.

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 default configuration files for RSpec.

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.

Exercises

- If you've got an existing Rails application that needs a test coverage, feel free to begin building a test suite now, following the steps we took to install and configure RSpec in TasteDrivenDishes.
- Or, try creating a new Rails app from scratch! It doesn't need to be fancy or even unique. A simple to-do list or blog is always great for learning, or perhaps a tool to help track items in your favorite collection. Be as creative as you'd like!
- *Or*, perhaps you have ideas for features you'd add to TasteDrivenDishes! If that's the case, take a little time to build them out. They don't need to be fancy or pretty, but you may want to do this work in a separate copy of the code to avoid conflicts in future chapters.

Whichever path you choose, don't worry about tests yet. If you took the existing Rails app or scratch app route, install and configure RSpec in it now. Make sure you've installed the necessary gems and configured for the application. Ensure `bin/rspec` successfully runs.

3. Model specs

With RSpec successfully installed, let's put it to work and begin building a suite of reliable tests. We'll get started with TasteDrivenDishes'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.

To follow along with the code in this chapter, start with the *02-setup* folder from the sample code. Refer to the *03-models* folder for the completed code from this chapter. See "Downloading the sample code" from Chapter 1 for download instructions.

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:

- When instantiated with valid attributes, an object of the model should be valid.
- Objects that don't meet validation requirements 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 simplest requirements:

```
describe User do
  it "is valid with an email, password, nickname, and API token"
  it "requires a nickname"
  it "requires a unique nickname"
  it "requires an email"
  it "requires a unique email"
  it "requires a password"
  it "requires an API token"
  it "sets a new user's API token"
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 each validation 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 requires an API token*, *User requires an email*, *User sets a new user's API token*. 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 creating the new file:

```
rails g rspec:model user
      createÂ  spec/models/user_spec.rb
```

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

spec/models/user_spec.rb

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

The new file gives us our first look at some RSpec syntax and conventions. First, we *require* the file *rails_helper* in this file, and will do so in pretty much every file in our test suite. This tells RSpec that we need the Rails application to load, so it can then 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 12, when we begin practice test-driven development. For now, happens if we run this, using `bin/rspec`?

```
User
  add some examples to (or delete) /Users/asumner/code/examples/recipes/spec/mode\
ls/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/recipes/s\
pec/models/user_spec.rb
    # Not yet implemented
    # ./spec/models/user_spec.rb:4

Finished in 0.00058 seconds (files took 0.66131 seconds to load)
1 example, 0 failures, 1 pending
```

> You don't need to use generators to create spec files, or *any* file in a Rails application, for that matter. 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

```
require 'rails_helper'

RSpec.describe User, type: :model do
  it "is valid with an email, password, nickname, and API token"
  it "requires a nickname"
  it "requires a unique nickname"
  it "requires an email"
  it "requires a unique email"
  it "requires a password"
  it "requires an API token"
  it "sets a new user's API token"
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 something like:

```
User
  is valid with an email, password, nickname, and API token (PENDING: Not yet imp\
lemented)
  requires a nickname (PENDING: Not yet implemented)
  requires a unique nickname (PENDING: Not yet implemented)
  requires an email (PENDING: Not yet implemented)
  requires a unique email (PENDING: Not yet implemented)
  requires a password (PENDING: Not yet implemented)
  requires an API token (PENDING: Not yet implemented)
  sets a new user's API token (PENDING: Not yet implemented)

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

  1) User is valid with an email, password, nickname, and API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:4

  2) User requires a nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:5

  3) User requires a unique nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:6

  4) User requires an email
    # Not yet implemented
    # ./spec/models/user_spec.rb:7

  5) User requires a unique email
```

```
    # Not yet implemented
    # ./spec/models/user_spec.rb:8

  6) User requires a password
    # Not yet implemented
    # ./spec/models/user_spec.rb:9

  7) User requires an API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:10

  8) User sets a new user's API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:11


Finished in 0.00086 seconds (files took 0.83556 seconds to load)
8 examples, 0 failures, 8 pending
```

Great! Eight 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. Instead of saying something `should` or `should_not` match expected output, you `expect` something `to` or `not_to` be something else.

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

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

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

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

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

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

spec/models/user_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    it "is valid with an email, password, nickname, and API token" do
5      user = User.new(
6        email: "test@example.com",
7        password: "password",
8        nickname: "test",
9        api_token: "token"
10     )
11
12     expect(user).to be_valid
13   end
14
15   it "requires a nickname"
16   it "requires a unique nickname"
17   it "requires an email"
18   it "requires a unique email"
19   it "requires a password"
20   it "requires an API token"
21   it "sets a new user's API token"
22 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:

```
User
  is valid with an email, password, nickname, and API token
  requires a nickname (PENDING: Not yet implemented)
  requires a unique nickname (PENDING: Not yet implemented)
  requires an email (PENDING: Not yet implemented)
  requires a unique email (PENDING: Not yet implemented)
  requires a password (PENDING: Not yet implemented)
  requires an API token (PENDING: Not yet implemented)
  sets a new user's API token (PENDING: Not yet implemented)

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

  1) User requires a nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:15

  2) User requires a unique nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:16

  3) User requires an email
    # Not yet implemented
    # ./spec/models/user_spec.rb:17

  4) User requires a unique email
    # Not yet implemented
    # ./spec/models/user_spec.rb:18

  5) User requires a password
    # Not yet implemented
    # ./spec/models/user_spec.rb:19

  6) User requires an API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:20

  7) User sets a new user's API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:21
```

```
Finished in 0.02298 seconds (files took 0.56166 seconds to load)
8 examples, 0 failures, 7 pending
```

Congratulations, you've written your first test!

Matchers are key components in RSpec, so let's take a moment to break down what `be_valid` implies. We could have written the test as

```
expect(user.valid?).to eq true
```

or

```
expect(user.valid?).to be true
```

In these variations, `eq` and `be` are also matchers. Matchers help make tests read like plain English, rather than the `assert`-style language you might have seen in other frameworks. `be_valid` has something else happening–before comparing the left side of the matcher to the right, it also calls the `valid?` method on the user under test. That's why we don't need to explicitly call `user.valid?` in the original test.

RSpec includes many built-in matchers; we'll explore some of them throughout the book. But for now, let's write some more tests!

Testing validations

Validations are great for getting comfortable with automated testing. These tests can usually be written in just a few lines of code. Let's fill in our `nickname` validation spec:

spec/models/user_spec.rb

```
1  it "requires a nickname" do
2    user = User.new(nickname: nil)
3
4    expect(user).to be_invalid
5  end
```

This time, we *expect* that a new user with the `nickname` attribute explicitly set nil will *not* be valid. In this case, the `be_invalid` matcher calls `user.invalid?` before comparing the expected and actual values.

That's great, but the test doesn't tell the ready *why*. Let's fix that:

spec/models/user_spec.rb

```
it "requires a nickname" do
  user = User.new(nickname: nil)

  expect(user).to be_invalid
  expect(user.errors[:nickname]).to include("can't be blank")
end
```

Now, the test clarifies the reason the user isn't valid, by ensuring the error message matches what we'd expect from this validation. We check for this using RSpec's `include` matcher, which checks to see if a value is contained within an enumerable value (here, `errors`). 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

```
it "requires a nickname" do
  user = User.new(nickname: nil)

  expect(user).to be_invalid
  expect(user.errors[:nickname]).to_not include("can't be blank")
end
```

And sure enough, RSpec reports a failure:

```
Failures:

  1) User requires a nickname
    Failure/Error: expect(user.errors[:nickname]).to_not include("can't be blank")
      expected ["can't be blank"] not to include "can't be blank"
    # ./spec/models/user_spec.rb:19:in `block (2 levels) in <top (required)>'

Finished in 0.02385 seconds (files took 0.55168 seconds to load)
8 examples, 1 failure, 6 pending

Failed examples:

rspec ./spec/models/user_spec.rb:15 # User requires a nickname
```

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 temporarily comment out the `nickname` validation:

app/models/user.rb

```
require "open-uri"

class User < ApplicationRecord
  include Clearance::User

  has_many :recipes, dependent: :destroy
  has_many :favorites, dependent: :destroy
  has_many :favorite_recipes, through: :favorites, source: :recipe
  has_many :comments, dependent: :destroy
  has_one_attached :avatar

  # validates :nickname, presence: true, uniqueness: true
  validates :api_token, presence: true, uniqueness: true

  before_validation :set_api_token, on: :create
  before_create :set_avatar

  # remainder of file omitted ...
```

Run the specs again. This time, you should again see a failure. We told RSpec that a user with no nickname 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 `:email` validation.

spec/models/user_spec.rb

```
it "requires an email" do
  user = User.new(email: nil)

  expect(user).to be_invalid
  expect(user.errors[:email]).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 nickname attribute:

spec/models/user_spec.rb

```
it "requires a unique nickname" do
  User.create(
    nickname: "test",
    email: "test1@example.com",
    password: "password"
  )

  user = User.new(
    nickname: "test"
  )

  expect(user).to be_invalid
  expect(user.errors[:nickname]).to include("has already been taken")
end
```

Notice a subtle difference here: In this case, we first persisted a user (calling `create` on `User` instead of `new`) to build our test data, 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 nickname, 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 the `Recipe` model.

Say we want to make sure that users can't give two of their recipes the same name–the name should be unique within the scope of that user. In other words, *I*

can't have two recipes named *Vegetable Stir Fry*, but you and I could *each* have our own project named *Vegetable Stir Fry*. How might you test that?

As we did for users, we'll start by creating a new spec file for the Recipe model:

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

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

spec/models/recipe_spec.rb

```
require 'rails_helper'

RSpec.describe Recipe, type: :model do
  it "does not allow duplicate recipe names per user" do
    user = User.create(
      nickname: "test-user",
      email:      "test-user@example.com",
      password:   "password"
    )

    category = Category.create(name: "Test Category")

    user.recipes.create(
      name: "Test Recipe",
      category: category
    )

    second_recipe = user.recipes.build(
      name: "Test Recipe",
    )

    expect(second_recipe).to_not be_valid
    expect(second_recipe.errors[:name]).to include("has already been taken")
  end

  it "allows two users to share a project name" do
    user = User.create(
      nickname: "test-user",
      email:      "test-user@example.com",
      password:   "password"
    )

    other_user = User.create(
      nickname: "another-test-user",
      email:      "another-test-user@example.com",
      password:   "password"
```

```
37      )
38
39      category = Category.create(name: "Test Category")
40
41      user.recipes.create(
42        name: "Test Recipe",
43        category: category
44      )
45
46      second_recipe = other_user.recipes.build(
47        name: "Test Recipe",
48        category: category
49      )
50
51      expect(second_recipe).to be_valid
52    end
53  end
```

This time, since the `User` and `Recipe` models are coupled via an Active Record relationship, as are the `Recipe` and `Category` models, we need to provide a little extra information. In the case of the first example, we've got a user to which both recipes are assigned. In the second, the same project name is assigned to two unique recipes, 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 though it's not part of what we're testing here, in order to fulfill the app's requirement that each recipe belong to a category, we also need to create a category in each of the tests.

Since the `Recipe` model has the following validation:

app/models/recipe.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. We've got the beginnings of a feature to treat new users differently than users who've been around for a little while. Perhaps someday, we might limit the number of recipes or comments a new user can post. For now, we're just displaying a badge when attributing a recipe to a new user.

To handle this, we've got this method in the User class:

app/models/user.rb

```
def new_to_site?
  created_at > 1.month.ago
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

```
1  it "indicates a new user" do
2    user = User.new(created_at: Time.now)
3
4    expect(user.new_to_site?).to be true
5  end
```

Cool, but what about a user who's been here for awhile? We should test that, too:

spec/models/user_spec.rb

```
1  it "indicates an established user" do
2    user = User.new(created_at: 1.month.ago)
3
4    expect(user.new_to_site?).to be false
5  end
```

Not too bad, but let's tidy these tests up with some matcher magic. Remember earlier in this chapter that we used `be_valid` and `be_invalid` as matchers for testing a User object's validity? We can use `be_` on our own methods that return booleans, too!

spec/models/user_spec.rb

```
it "indicates a new user" do
  user = User.new(created_at: Time.now)

  expect(user).to be_new_to_site
end

it "indicates an established user" do
  user = User.new(created_at: 1.month.ago)

  expect(user).to_not be_new_to_site
end
```

Personally, I love this feature of RSpec. I think it makes tests read more like documentation and less like code. If you or your team disagree, there's nothing wrong with the previous iterations of these tests.

Either way, we're establishing a pattern for testing: Create test data, then tell RSpec how you expect it to behave. Let's keep going.

Testing class methods and scopes

Our users can search recipe titles for a provided term. For the sake of demonstration, it's currently implemented as a scope on the Recipe model:

app/models/recipe.rb

```
scope :by_word_in_name, ->(query) {
  where("name LIKE ?", "%#{query}%") if query.present?
}
```

Let's add another test to `recipe_spec` to cover this:

spec/models/recipe_spec.rb

```
it "finds recipes that contain the search term in their name" do
  user = User.create(
    nickname: "test-user",
    email: "test-user@example.com",
    password: "password"
  )

  category = Category.create(name: "Test Category")

  first_recipe = user.recipes.create(
    name: "Pepperoni Pizza",
    category: category
  )

  second_recipe = user.recipes.create(
    name: "Cheese Pizza",
    category: category
  )

  results = Recipe.by_word_in_name("pepperoni")

  expect(results).to include(first_recipe)
  expect(results).to_not include(second_recipe)
end
```

The `by_word_in_name` scope should return a collection of recipes matching the search term, and that collection should only include those recipes–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 recipes containing the search term?

Testing all the cases

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/recipe_spec.rb

```
it "returns an empty collection when no recipes matching the search term are foun\
d" do
  user = User.create(
    nickname: "test-user",
    email: "test-user@example.com",
    password: "password"
  )

  category = Category.create(name: "Test Category")

  first_recipe = user.recipes.create(
    name: "Pepperoni Pizza",
    category: category
  )

  second_recipe = user.recipes.create(
    name: "Cheese Pizza",
    category: category
  )

  results = Recipe.by_word_in_name("veggie")

  expect(results).to be_empty
end
```

This spec checks the value returned by `Recipe.by_word_in_name("veggie")`. Since the resulting collection *is* empty, the spec passes! We're testing not just for the ideal 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. 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.

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. Try to only check for one result per example. (We'll talk about exceptions to this in a later chapter.)
- **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.)

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

Exercises

If you're following along with your own, untested Rails code, take a look at its models now. What attributes map to their underlying database tables? How are data validated? What other business logic do they contain?

These are all great candidates for your first test coverage. Start by generating a model spec for a given model, then outline the scenarios you want to test. Think through how to build just enough test data for each scenario, and how it would prove that your code behaves as you want. Then write each spec and run. I recommend doing this one test at a time. Don't worry if you're repeating yourself from test to test. We'll explore de-duplication options in the next chapter.

As you're writing and running you new specs, do you notice anything unexpected about your application code? Like, a feature that doesn't quite work as you'd assumed it did, or code that would benefit from refactoring? Congratulations! You're already seeing the benefits of test-driven software design!

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. Visit Everyday Rails at https://everydayrails.com/.

About the author

Aaron Sumner is a software engineer with nearly 35 years working professionally, developing web applications in the education and publishing industries. Since 2005, he has worked primarily in Ruby on Rails, though in his day job as an Engineering Manager at O'Reilly Media, spends time in Python and Go. He enjoys helping small development teams achieve ambitious goals through the pragmatic use of technology.

Aaron resides in Saint Louis, Missouri, with his wife, Caitlin, and dog, Lieutenant Dan. Away from work he enjoys woodworking and being outside.

Visit Aaron's personal site at https://aaronsumner.com/.

Colophon

The original cover image “Old truck in early morning light stock photo” is by iStockphoto contributor Habman_18.