THE BEGINNER'S GUIDE TO RAILS TESTING

JASON SWETT

host of the
Rails with Jason Podcast

The Beginner’s Guide to Rails Testing

Jason Swett

Contents

Introduction

Welcome to *The Beginner's Guide to Rails Testing*! The purpose of this guide is to answer your biggest testing questions and help you get properly oriented as you begin your journey toward competent Ruby on Rails testing.

The questions answered in this book are questions I've encountered over roughly three years of teaching Rails testing. As I've written blog posts, interacted on forums, participated in chat rooms, and had email exchanges, I've had the opportunity to hear from dozens if not hundreds of Rails developers trying to learn testing. The same questions tend to come up over and over, which are the eight questions I've included in this guide.

I hope this guide helps you. If you read it and still have unanswered questions, please email me at jason@codewithjason.com and I'll do my best to help you.

Good luck and thanks for reading.

Chapter 1

What are the different kinds of Rails tests and when should I use each?

When starting out with Rails testing, it's hard to know where to start.

First, there's the decision of which framework to use. Then, if you've chosen RSpec (which most people do), you're presented with a bewildering set of possible test types to use.

In this chapter I'll show you what types of tests there are. I'll show you which ones you should use and which ones you can ignore. Since most commercial Rails projects use RSpec, I'm going to focus on the eight types of tests that the RSpec library offers. (Although if I were to use Minitest, my strategy regarding test types would be pretty much the same.)

1.1 The eight types of RSpec specs

The RSpec library offers a lot of different spec types.

- Model specs
- System specs/feature specs*

- Request specs/controller specs*
- Helper specs
- View specs
- Routing specs
- Mailer specs
- Job specs

There are two lines with asterisks. These are cases where the RSpec team decreed one spec type obsolete and replaced it with a new type. I'm only including those ones for completeness.

So the up-to-date list is really the following.

- Model specs
- System specs
- Request specs
- Helper specs
- View specs
- Routing specs
- Mailer specs
- Job specs

Here's when I use each.

- Model specs - **always**
- System specs - **always**

- Request specs - **rarely**
- Helper specs - **rarely**
- View specs - **never**
- Routing specs - **never**
- Mailer specs - **never**
- Job specs - **never**

Let's talk about each of these spec types in detail. I'll explain why I use the ones I use and why I ignore the ones I ignore.

1.2 Spec types I always use

Believe it or not, the overwhelming majority of the Rails tests I write make use of just two of the eight different spec types offered by RSpec. You might think that this would leave large gaps in my test coverage but it doesn't. My test coverage is consistently above 95%.

1.2.1 System specs

System specs are "high-level" tests that simulate a user's keystrokes and mouse clicks. System specs literally open up a browser window (although perhaps an invisible browser window if the tests are run "headlessly") and use certain tools to manipulate the browser to exercise your application through simulated user input.

The reason I find system specs so valuable is that they test my whole stack, not just a slice of it, and they test my application in the same exact way that a real user will be using it. System specs are the only type of test that give me confidence my whole application really works.

I write so many system specs that I've developed a repeatable formula for adding system specs to any new CRUD feature.

1.2.2 Model specs

Even though system specs are indispensable, they're not without drawbacks. System specs are somewhat "heavy". They're often more work to write and more expensive to run than other types of tests. For this reason I like to cover my features with a small number of coarse-grained system specs and a comparatively large number of fine-grained model specs.

As the name implies, model specs are for testing models. I tend to only bring model specs into the picture once a model has reached a certain level of "maturity". At the beginning of a model's life, it might have all its needs covered by built-in Rails functionality and not need any methods of its own. Some people write tests for things like associations and validations but I don't because I find those types of tests to be pointless.

I use model specs to test my models' methods. When I do so, I tend to use a test-first approach and write a failing test before I add a new line of code so that I'm sure every bit of code in my model is covered by a test.

1.3 Spec types I rarely use

1.3.1 Request specs

Request specs are more or less a way to test controller actions in isolation. I tend not to use request specs much because in most cases they would be redundant to my system specs. If I have system specs covering all my features, then of course a broken controller would fail one or more of my tests, making tests specifically for my controllers unnecessary.

I also try to keep my controllers sufficiently simple as to not call for tests of their own.

There are just three scenarios in which I do use request specs. First: If I'm working on a legacy project with fat controllers, sometimes I'll use request specs to help me harness and refactor all that controller code. Second: If I'm working on an API-only Rails app, then system specs are physically impossible and I drop down to request specs instead. Lastly, if it's just too awkward or expensive to use a system spec in a certain case then I'll use a request spec instead. I write more about my reasoning here.

1.3.2 Helper specs

The reason I rarely write helper specs is simple: I rarely write helpers.

1.4 Spec types I never use

1.4.1 View specs and routing specs

I find view specs and routing specs to be redundant to system specs. If something is wrong with one of my views or routes, it's highly likely that one of my system specs will catch the problem.

1.4.2 Mailer specs and job specs

I don't write mailer specs or job specs because I try very hard to make all my mailers and background jobs one-liners (or close). I don't think mailers and background jobs should *do* things, I think they should only *call* things. This is because mailers and background jobs are mechanical devices, not code organization devices.

To test my mailers and background jobs, I put their code into a PORO model and write tests for that PORO.

1.5 Takeaways

RSpec offers a lot of different spec types but you can typically meet 98% of your needs with just system specs and model specs.

If you're a total beginner, I'd suggest starting with system specs.

Chapter 2

What are all the Rails testing tools and how do I use them?

One of the most common questions for Rails developers new to testing is "What are all the Rails testing tools and how do I use them?"

I'll explain what the major tools are but I want to preface it by saying that the most important thing to learn to be a successful tester is testing principles, not testing tools. If you think of testing like a taco, the tools are the tortilla and the principles are the stuff inside the taco. The tortilla is essential but it's really only a vehicle.

The following are the tools I use for my testing.

2.1 RSpec

RSpec is a test framework. A test framework is what gives us a structure for writing our tests as well as the ability to run our tests.

There are other test frameworks but RSpec is the most popular one for commercial Rails projects. The second most popular test framework is Minitest.

Test frameworks differ syntactically but the testing principles and practices are going to be pretty much the same no matter what framework you're using. (If you're not sure whether you should learn RSpec or Minitest, I write about that here.)

2.2 Factory Bot

One of the challenges of Rails testing is generating test data. For example, if you're writing a test that logs a user in and then takes some action, you're going to have to create a user in the database at the beginning of the test. Many tests require much more complicated test data setup.

There are two common ways of generating test data in Rails tests: fixtures and factories.

2.2.1 Fixtures

Fixtures typically take the form of one or more YAML files with some hard-coded data. The data is translated into database records one time, before any of the tests are run, and then deleted afterward. (This happens in a separate test database instance of course.)

2.2.2 Factories

With factories, database data is generated specifically for each test. Instead of loading all the data once at the beginning and deleting it at the end, data is inserted before each test case and then deleted before the next test case starts. (More precisely, the data isn't deleted, but rather the test is run inside a database transaction and the data is never committed in the first place, but that's a mechanical detail that's not important right now.)

2.2.3 Relative merits of fixtures and factories

I tend to prefer factories because I like having my data generation right inside my test, close to where the test is happening. With fixtures the data setup is too distant from where the test happens.

In my experience, for whatever reason, most people who use RSpec use factories and most people who use Minitest use fixtures. If you'd like to learn more about factories and fixtures, I write more about it here.

2.3 Capybara

Some Rails tests only exercise Ruby code. Other tests actually open up a browser and simulate user clicks and keystrokes.

Simulating user input this way requires us to use some sort of tool to manipulate the browser. Capybara is a library that uses Ruby to wrap a *driver* (usually the Selenium driver), letting us simulate clicks and keystrokes using convenient Ruby methods.

For more examples of how to use Capybara, go here.

2.4 VCR and WebMock

One principle of testing is that tests should be *deterministic*, meaning they run the same way every time no matter what.

When an application's behavior depends on external services (e.g. a third-party API like Stripe) it makes it harder to have deterministic tests. The tests can be made to fail by an internet connection failure or a temporary outage of the external service.

Tools like VCR and WebMock can help smooth out these challenges. VCR can let us run our tests against the real external service, but capture all the service's responses in local files so that subsequent test runs don't talk to the external service but rather just go off of the saved responses. That way, even if the internet connection fails or the service goes down, the tests still work.

WebMock is a tool that serves a similar purpose, although I usually use it in a more limited way. I don't consider my test suite to be deterministic unless it doesn't talk to the network at all, so I use WebMock to enforce that my test suite isn't making any network requests.

2.5 Takeaways

Rails testing tools take some time to learn, but the important part (and perhaps more difficult part) is learning testing principles.

If you’re just getting started with Rails testing, the next step I would suggest is to learn about the different types of Rails tests and when to use them.

Chapter 3

Which test framework should I learn, RSpec or Minitest?

A common Rails testing question is which testing framework to use. RSpec and Minitest are the two that most people are deciding between. To many beginners it's not clear which is the better choice.

We could weigh the technical pros and cons of each framework. Many people find things to love and hate about both RSpec and Minitest. You can find some passionate flame wars online if you look.

But before we get into all that, there are some realities to consider that overshadow the relative technical merits of the two frameworks. There are two particular facts we should think about.

3.1 Fact #1: usually, someone else decides for you

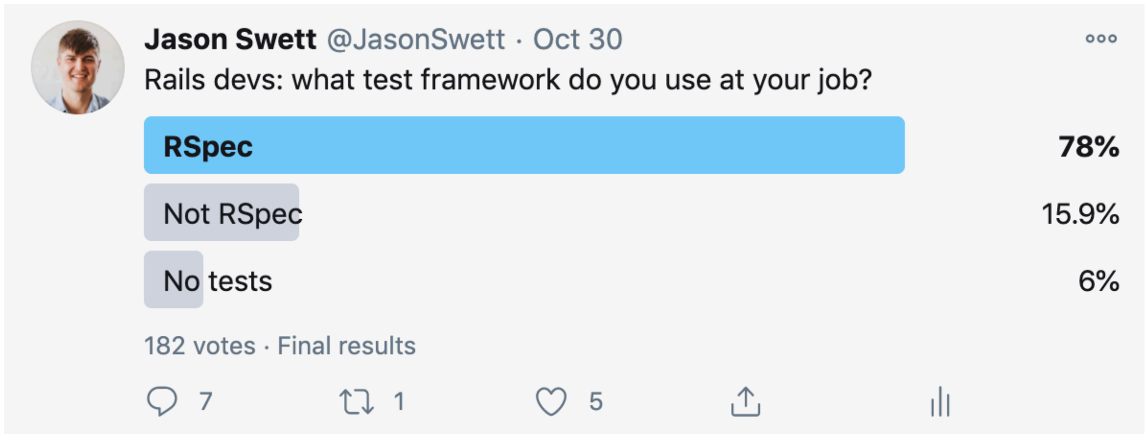
Most of us don't have much choice as to whether to use RSpec or Minitest at work.

At some point we'll get a job. At that job they'll either use RSpec there or Minitest (or something else or nothing at all). Whatever they use at work, that's what we'll be using. Our personal preferences are moot.

3.2 Fact #2: usually, they've chosen RSpec

For better or worse, it's my experience and the experience of most Rails developers I've talked with that most commercial projects use RSpec. (Note how I said most *commerical* projects. Most commercial projects use RSpec and most OSS Ruby projects, in my experience, use Minitest. I do not know why this is the way it is.)

Out of curiosity I did a (totally unscientific) poll regarding which test framework they use at work. Take it with a grain of salt, but here are the results.


Jason Swett @JasonSwett · Oct 30
Rails devs: what test framework do you use at your job?

RSpec	**78%**
Not RSpec	15.9%
No tests	6%

182 votes · Final results

7 1 5

Even if my numbers are off by quite a bit, RSpec is still the more popular framework.

3.3 What does this mean?

My take is that this means if your goal is to get a Rails job, learning RSpec over Minitest will give you a higher probability that your skills match the tech stack that's used at any particular company.

Some people may object to this way of looking at it. They might argue that if you always you go with whatever's most popular instead of what's the best technical choice, you may end up using a Windows laptop or switching from Rails to Node.js.

This argument is flawed though. We're free to make our own choices on the big things but we can't dictate what comes along with those choices. We can choose to use Rails instead of a different framework, but we can't reasonably say that we're only going to work on Rails projects that use, for example, Minitest and MySQL and Angular and no other combination of technologies. We have to compromise a little or face extremely limited job options.

3.4 Also, it doesn't matter much

Having said all that, I actually don't believe your choice of which test framework to learn matters!

RSpec and Minitest differ syntactically but they don't really have meaningful conceptual differences. The principles of testing are the same regardless of which test framework you're using, or even which language you're using for that matter.

You're very unlikely to become an expert in Minitest and then get turned down for a job because they use RSpec there, or vice versa. Employers typically realize that if someone is skilled with testing, they'll be able to pick up any test framework relatively easily.

3.5 So try both

In a sense it might sound depressing that the answer to the RSpec/Minitest question is a) we don't have a choice and b) it doesn't matter anyway. I actually find these facts freeing.

If the choice between RSpec and Minitest doesn't matter that much then we're free to evaluate both according to our own independent judgment and taste and not worry about whether we're making the "right" choice. Whatever we choose, we're likely to develop skills that will apply to any job, whether they use Minitest or RSpec there.

So my advice is to try both frameworks and see which one you like better. Neither one is objectively superior to the other.

3.6 But if you just want me to pick for you, I say RSpec

My very simplistic logic is that RSpec is what you'll most likely be forced to use at work, so that's what you might as well learn.

But again, I encourage you to try both and decide for yourself. This is ultimately not a very important decision. Learning testing principles is much more important than learning testing frameworks.

Chapter 4

How do I make testing a habitual part of my development work?

One of the most common questions asked by Rails developers new to testing is: how do I make testing a habitual part of my development work?

It's one thing to know how to write tests. It's another thing to actually write tests consistently as a normal part of your work.

In order to share with you how to make testing a habitual part of your development work, I conducted a poll among some of my peers in the Rails world to see what keeps them in the habit of writing tests consistently. I also examined my own motivations.

When I drew the commonalities among the answers, what I came up with was a trifecta not unlike Larry Wall's three virtues of a great programmer. The trifecta is **laziness**, **fear**, and **pride**. Let's examine each "virtue" individually.

4.1 Laziness

It might sound funny to name laziness as the first motivation for writing tests habitually. After all, tests seem like *extra* work. Writing tests consistently seems like something that would require *discipline*. But for me and many of the people who responded to my poll, it's quite the opposite.

4.1.1 The alternative to automated tests

The alternative to writing tests isn't just doing nothing. The alternative to writing tests is to perform manual testing, to let your users test your application for you in production, or most likely, a combination of the two. The alternative to writing tests is to suffer great pain and toil.

The laziness factor also extends beyond QA. I personally find that the process of writing features is often easier and more pleasant when I'm writing with the assistance of tests than when I'm not.

4.1.2 Mental energy

Mental energy is a finite, precious resource that (for me at least) starts full in the morning and depletes throughout the day. When I'm working I don't ever want to use more than the minimum amount of mental exertion necessary to complete a task.

If I write a feature without using tests, I'm often juggling the "deciding what to do" work and the "actually doing it" work at the same time, which has a cognitive cost more than twice as much as performing those two jobs separately in serial. When I build a feature with the aid of tests, the tests allow me to separate the "deciding what to do" work from the "actually doing it" work.

It works like this. First I capture what to do in the form of a test. Then I follow my own instructions by getting the test to pass. Then I repeat. This is a much lighter cognitive burden than if I were to juggle these different mental jobs and allows me to be productive for longer because I don't run out of mental energy as early in the day.

4.1.3 Code understandability

It’s more difficult, time-consuming and unpleasant to work with messy code than to work with clear and tidy code.

Being a lazy person, difficult, time-consuming and unpleasant work is exactly what I *don’t* want to do. I want to do work that’s pleasant, quick and easy.

Unfortunately it’s not possible to have clean, understandable code without having automated tests. This might sound like a hyperbolic claim but it’s not. I can prove it based on a chain of truths.

The first truth is that it’s impossible to write a piece of code cleanly on the first try. Some amount of refactoring, typically a lot of refactoring, is necessary in order to get the code into a reasonably good state. This is true on a feature-by-feature basis but it’s especially true on the scale of a whole project codebase.

The second truth is that it’s impossible to do non-trivial refactorings without having automated tests. The feedback cycle is just too long when all the testing is done manually. Either that or the risk of refactoring without testing afterward is just too large to be justified.

So, if it’s impossible to have good code without refactoring, and it’s impossible to do refactoring without tests, then it’s impossible to have good code without tests.

My extreme personal laziness demands that I only write neat and understandable code. Therefore, I have to write tests in order to satisfy my laziness.

4.2 Fear

Fear is another powerful impetus for testing. If I don’t write tests for my features, it increases the risk that I release a bug to production. Bugs cause me shame and embarrassment. I don’t want to feel embarrassment or shame.

Bugs may also have negative business consequences to the company I work for. This could negatively affect the company’s ability or willingness to pay me as much as I want.

When laziness doesn’t drive me to write tests, fear often does.

4.3 Pride

Lastly there's pride. (I find Larry Wall's "hubris" a little too strong a word.)

Sometimes, when I'm tempted not to write a test for a feature, I imagine another developer stumbling across my work in the future and seeing that there are no tests. I imagine myself sheepishly admitting to that developer that I didn't bother to write tests for that feature. Why didn't I write tests? No good reason.

As the arrogant person that I am, this imaginary interaction brings me pain. I really don't like the idea that somebody else would like at my work and make a (legitimate) negative judgment.

I also want my work to be exemplary. If we hire a junior developer where I work, I want to be able to point to my code and say "*This* is how we do it." I don't know how I would explain that my test coverage is poor but I want theirs to be good.

4.4 Takeaways

I'm not driven to write tests out of discipline. I also don't consider testing to be "extra" effort but rather an effort-saver.

The main forces that drive me to write tests are laziness, fear and pride. Mostly laziness.

Chapter 5

What level of test coverage should I shoot for?

“What level of test coverage should I shoot for?” is one of the questions most commonly asked by beginners to Rails testing.

My answer is that you shouldn’t shoot for a particular level of test coverage. I recommend that instead you make testing a habitual part of your development workflow. A healthy level of test coverage will flow from there.

I also want to address *why* people ask this question. I think people ask this because they want some way of knowing whether they’re testing “enough” or doing testing “right”. Test coverage is one way of measuring this but I think there are better, more meaningful ways.

I think that if you’re feeling the kinds of pains that missing tests leave in their absence, then you need more tests. If you’re not feeling those kinds of pains, then you’re good.

5.1 Pains that tell you your test coverage might be insufficient

5.1.1 Too many bugs

This is the obvious one. All software has bugs, but if you feel like the rate of new bugs appearing in production is unacceptably high, it may be a symptom of too little test coverage.

5.1.2 Too much manual testing

This is another fairly obvious one. The only alternative to using automated tests, aside from not testing at all, is to test manually.

Some level of manual testing is completely appropriate. Automated tests can never replace, for example, exploratory testing done by a human. But humans should only carry out the testing that can't be done better by a computer. Otherwise testing is much more expensive and time-consuming than it needs to be.

5.1.3 Infrequent deployments

Infrequent deployments can arise as a symptom of too few tests for a couple different reasons.

One possible reason is that the need for manual testing bottlenecks the deployment timing. If it takes two days for manual testers to do a full regression test on the application, you can of course only deploy a fully-tested version of your application once every two days at maximum. (And this is assuming the test suite passes every time, which is not typically the case.)

Another possible reason for infrequent deployments is the following logic: things go wrong every time we deploy, therefore things will go wrong less often if we deploy less often, so let's deploy less often. Unfortunately this decision means that problems pile up and get introduced to production all at once on each deployment instead of getting sprinkled lightly over time.

With the presence of a good test suite, deployments can happen many times a day instead of just once every few weeks or months.

5.1.4 Inability to refactor or make big changes

When a particular change has a small footprint, manual testing is usually good enough (although of course sometimes changes that seem like they'd have small footprints cause surprising regressions in distant areas).

When a change has a large footprint, like a Rails version upgrade or a broad refactoring, it's basically impossible to gain sufficient confidence of the safety of the change without having a solid automated test suite. So on codebases without good test coverage, these types of improvements tend not to happen.

5.1.5 Poor code quality

As I've written elsewhere, it's not possible to have clean, understandable code without having tests.

The reason is that refactoring is required in order to have good code and automated tests are required in order to do sufficient refactoring.

5.1.6 Diminished ability to hire and retain talent

Lastly, it can be hard to attract and retain high-quality developers if you lack tests and you're suffering from the ailments that result from having poor test coverage.

If a job candidate asks detailed questions about your development practices or the state of your codebase, he or she might develop a negative perception of your organization relative to the other organizations where he or she is interviewing. All other things being equal, a sophisticated and experienced engineer is probably more likely to pick some other organization that does write tests over yours which doesn't.

Even if you manage to get good people on your team, you might have trouble keeping them. It's painful to live with all the consequences of not having tests. Your smartest people are likely to be the most sensitive to these pains, and they may well seek somewhere else to work where the development experience is more pleasant.

5.2 The takeaway

I don't think test coverage is a particularly meaningful way to tell whether you're testing enough. Instead, assess the degree to which you're suffering from the above symptoms of not having enough tests. Your degree of suffering is probably proportionate to your need for more tests.

If you do this, then "good" coverage numbers are likely to follow. Last time I checked my main codebase at work my test coverage level was 96.47%.

Chapter 6

How do I set up a new Rails project for testing?

Below is how I set up a fresh Rails application for testing. I'll describe it in three parts:

1. An application template that can add all the necessary gems and configuration
2. My setup process (commands I run to create a new Rails app)
3. A breakdown of the gems I use

Let's start with the application template.

6.1 My application template

First, if you don't know, it's possible to create a file called an application template that you can use to create a Rails application with certain code or configuration included. This is useful if you create a lot of new Rails applications with parts in common.

Here's an application template I created that will do two things: 1) install a handful of testing-related gems and 2) add a config file that will tell RSpec not to generate certain types of files. A more detailed explanation can be found below the code.

Listing 6.1: Gemfile

```
gem_group :development, :test do
  gem 'rspec-rails'
  gem 'factory_bot_rails'
  gem 'capybara'
  gem 'webdrivers'
  gem 'faker'
end

initializer 'generators.rb', <<-CODE
  Rails.application.config.generators do |g|
    g.test_framework :rspec,
      fixtures:         false,
      view_specs:       false,
      helper_specs:     false,
      routing_specs:    false,
      request_specs:    false,
      controller_specs: false
  end
CODE
```

The first chunk of code will add a certain set of gems to my **`Gemfile`**. A more detailed explanation of these gems is below.

The second chunk of code creates a file at **`config/initializers/generators.rb`**. The code in the file says "when a scaffold is generated, don't generate files for fixtures, view specs, helper specs, routing specs, request specs or controller specs". There are certain kinds of tests I tend not to write and I don't want to clutter up my codebase with a bunch of empty files. That's not to say I *never* write *any* of these types of tests, just sufficiently rarely that it makes more sense for me to create files manually in those cases than for me to allow files to get generated every single time I generate a scaffold.

6.2 The setup process

When I run **`rails new`**, I always use the **`-T`** flag for "skip test files" because I always use RSpec instead of the Minitest that Rails comes with by default.

Also, incidentally, I always use PostgreSQL. This choice of course has little to do with testing but I'm including it for completeness.

In this particular case I'm also using the **`-m`** flag so I can pass in my application template. Application templates can be specified using either a local file path or a URL. In this case I'm using a URL so that you can just copy and paste my full **`rails new`** command as-is if you want to.

```
$ rails new my_project -T -d postgresql \
  -m https://raw.githubusercontent.com/jasonswett/testing_application_template/master/applicati
```

Once I've created my project, I add it to version control. (I could have configured my application template to do this step manually, but I wanted to explicitly show it as a separate step, partially to keep the application template clean and easily understandable.)

```
$ git add .
$ git commit -a -m'Initial commit'
```

Lastly, I run **`rails g rspec:install`**. Even though I've already added the RSpec gem via my application template, I need to run **`rails g rspec:install`** in order to add some RSpec-related configuration files to my project.

```
$ rails g rspec:install
```

6.3 The gems

Here's an explanation of each gem I chose to add to my project.

6.3.1 rspec-rails

RSpec is one of the two most popular test frameworks for Rails, the other being Minitest.

The **`rspec-rails`** gem is the version of the RSpec gem that's specifically fitted to Rails.

6.3.2 factory_bot_rails

Factory Bot is a tool for generating test data. Most Rails projects that use RSpec also use Factory Bot.

Like **rspec-rails**, **factory_bot_rails** is a Rails-specific version of a more general gem, factory_bot.

6.3.3 capybara

Capybara is a tool for writing *acceptance tests*, i.e. tests that interact with the browser and simulate clicks and keystrokes.

The underlying tool that allows us to simulate user input in the browser is called Selenium. Capybara allows us to control Selenium using Ruby.

6.3.4 webdrivers

In order for Selenium to work with a browser, Selenium needs *drivers*. There are drivers for Chrome, drivers for Edge, etc. Unfortunately it can be somewhat tedious to keep the drivers up to date. The webdrivers gem helps with this.

6.3.5 faker

By default, Factory Bot (the tool for generating test data) will give us *factories* that look something like this:

```ruby
FactoryBot.define do
  factory :customer do
    first_name { "MyString" }
    last_name { "MyString" }
    email { "MyString" }
  end
end
```

This is fine for just one record but becomes a problem if we have multiple records plus a unique constraint. If in this example we require each customer to have a unique email address, then we'll get a database error when we create two **customer** records because the email address of **MyString** will be a duplicate.

One possible solution to this problem is to replace the instances of **"MyString"** with something like **SecureRandom.hex**. I don't like this, though, because I often find it helpful if my test values resemble the kinds of values they're standing in for. With Faker, I can do something like this:

```
FactoryBot.define do
  factory :customer do
    first_name { Faker::Name.first_name }
    last_name { Faker::Name.last_name }
    email { Faker::Internet.email }
  end
end
```

This can make test problems easier to troubleshoot than when test values are simply random strings like **c1f83cef2d1f74f77b88c9740cfb3c1e**.

6.3.6 Honorable mention

I also often end up adding the VCR and WebMock gems when I need to test functionality that makes external network requests. But in general I don't believe in adding code or libraries speculatively. I only add something once I'm sure I need it. So I typically don't include VCR or WebMock in a project from the very beginning.

6.4 Next steps

After I initialize my Rails app, I usually create a walking skeleton by deploying my application to a production and staging environment and adding one small feature, for example the ability to sign in. Building the sign-in feature will prompt me to write my first tests. By working in this way I front-load all the difficult and mysterious work of the project's early life so that from that point on, my development work is mostly just incremental.

If you're brand new to Rails testing and would like to see an example of how I would actually write a test once I have the above application set up, I might recommend my Rails testing "hello world" post.

Chapter 7

How do I add tests to an existing Rails project?

One of the most common questions asked by developers new to Rails testing is "How do I add tests to an existing Rails project?"

The answer largely depends on your experience level with testing. Here are my answers based on whether you have little testing experience or if you're already decently comfortable with testing.

7.1 If you have little testing experience

If you have little testing experience, I would suggest getting some practice on a fresh Rails app before trying to introduce testing to the existing Rails project you want to add tests to.

Adding tests to an existing project is a distinct skill from writing tests for new projects. Adding tests to an existing project can be difficult even for very experienced testers, for reasons described below.

At the same time, you probably don't want to wait a year to learn testing before you start enjoying the benefits of testing on your existing Rails app. What I would suggest is to first start a fresh throwaway Rails app for the purpose of learning testing. Then, once you've gotten a little experience there, see if you can apply something to your existing Rails app. Then, if things get too hard in the existing app, switch back to the throwaway app so you can strengthen your skills more. Continue switching back and forth until you don't need to anymore.

7.2 If you're already comfortable with testing

Here's how I suggest adding tests to an existing Rails project: 1) develop a shared vision with your team, 2) start with what's easiest, then 3) expand your test coverage.

7.2.1 Develop a shared vision

Going from no tests to decent test coverage is unfortunately not as simple as just deciding one day that from now on we're going to write tests.

The team maintaining the codebase needs to decide certain things, like what testing tools they're going to use and what testing approach they're going to use.

In other words, if the team wants to go from point A to point B, they have to decide exactly where point B is and how they intend to try to get there.

7.2.2 Start with what's easiest

When adding tests to a codebase that has few or no tests, it might seem logical to start by adding tests where tests would be most valuable. Or it might seem logical to require all new changes to have tests. Unfortunately, both these ideas have problems.

The features in an application that are most valuable are also likely to be among the most non-trivial. This means that tests for these features will probably be relatively hard to write due to the large amount of setup data needed. Code written without testability in mind can also be difficult to test due to entangled dependencies.

Requiring all new changes to have tests also has problems. New changes aren't usually independent of existing code. They're usually quite tangled up. This brings us back to the same problem we'd have adding tests to our most important features: the setup and dependencies make adding tests difficult, sometimes prohibitively so.

What I would do instead is start with what's easiest. I would look for the simplest CRUD interfaces in the app and add some tests there, even if those particular tests didn't seem to add much value. The idea isn't to add valuable tests right from the start but to establish a beachhead that can be expanded upon.

7.2.3 Expand

Once you have a handful of tests for trivial features, you can add tests for increasingly complicated features. This will give you a much better shot at ending up with good test coverage than trying to start with the most valuable features or trying to add tests for all new changes.

7.3 The mechanical details

If your existing Rails application doesn't have any testing infrastructure, I would suggest taking a look at my how I set up a Rails application post. (Remember that it's possible to apply an application template to an existing project.)

As you add tests to your project starting with the most trivial features, I would suggest starting with system specs as opposed to model specs or any other type of specs. The reason is that system specs are often more straightforward to conceive of and understand. If you'd like a formula you can apply to add system specs to almost any CRUD feature, you can find that here.

Then, as you get deeper into adding tests to your application, I would suggest two resources: Working Effectively with Legacy Code by Michael Feathers and my post about using tests as a tool to wrangle legacy projects. You might not consider your project a legacy project, but the techniques will be useful anyway.

Chapter 8

Should I be doing test-driven development?

When I see questions from beginners regarding learning testing, sometimes they seem to conflate testing with test-driven development (TDD). People will say "I have such-and-such question about TDD" but really it's just a question about testing, nothing to do with TDD specifically.

Other people sometimes ask questions about whether TDD is "better" than writing tests after.

In this chatper I'll try to clarify what's TDD and what's not. I'll also explain whether I think it makes sense for testing beginners to try to practice TDD.

8.1 Testing != TDD

First of all, at the risk of stating the obvious, testing and TDD aren't the same thing. TDD is a specific kind of testing practice where you write the tests before you write the code that makes the test pass. (If you want to go deeper into TDD, I highly recommend Kent Beck's Test Driven Development: By Example.)

8.2 Learning vs. incorporating

Another mistake beginners sometimes make is to conflate learning testing with incorporating testing as a habitual part of their development workflow. They feel like they need to start adopting testing practices into their workflow from day one, and if they fail to do that, then they've failed at learning testing.

I think it's more productive to separate the jobs of *learning* testing and *applying* testing. It's not like skiing, where you learn it and do it at the same time. It's more like basketball, where you practice free throws in your driveway and build some skills that way before you try to play a real game in front of an audience. You'll get farther in the beginning if you separate the practice from the application of what you've learned to production tests. When you get comfortable enough, you can take off the training wheels and get all your practice from writing production tests.

8.3 TDD is beneficial but optional

TDD is super helpful in certain scenarios but it's not something you absolutely need to learn when you're first learning testing. I think it's completely appropriate to first learn the fundamentals of testing in general, and then start to learn TDD once you've developed a decent level of comfort with testing.

8.4 I don't always practice TDD

I'm not an advocate of practicing TDD 100% of the time in Rails, even for experienced testers. The reason is that when I'm building a new feature, I often have little idea what shape that feature will take, and the most realistic way for me to hammer it into shape is to just start building it. Once I've built some of the feature, then I'll start adding tests. So, a portion of the time, I write my tests after writing my application code. The place where I find TDD most useful is for model code. I practice TDD in my models a high percentage of the time. Once I've put the broad strokes of a feature in place, I'll usually use TDD to work out the fine-grained aspects of it.

8.5 Takeaways

- Testing and test-driven development aren’t the same thing.
- When you’re first learning testing, it can be helpful to separate learning testing from applying testing.
- You don’t need to learn TDD when you’re starting out.
- I don’t always practice TDD or even advocate practicing TDD 100% of the time. I myself practice TDD maybe 60% of the time.