7 Tips for Faster

Rails Tests

Chris Kottom

The Minitest Cookbook

Slow tests are a drag.

And I don't just mean that figuratively. Writing code should be an exercise in building cathedrals of pure thought-stuff, but slow tests can suck the joy right out of it.

If tests are an integral part of your development workflow (and they should be), whether you're practicing by-the-book TDD or not, you should be running tests every few minutes at most. But if running your suite is slow enough that you have time to make coffee or flip over to Twitter or get distracted in any other way, then you've got a problem. If your tests are painful or getting in the way of you shipping, then it's time to start looking at ways to speed them up.

The tips contained in this mini-report are intended to provide a beginner-level introduction and a sanity check to get you started on rehabilitating your slow Rails test suite. If you find them useful, I'd love it if you would drop me a line and let me know.

Thanks for downloading!

Chris Kottom

chris@chriskottom.com

1. Use lightweight testing frameworks.

RSpec is still considered the most popular unit testing framework for Ruby, but at four separate gems and around 12,000 lines of code, there's no denying that it is beefy. Minitest, on the other hand, weighs in at a lean and mean 1600 LOC, and that combined with a limited scope and a philosophy that avoids trying to be all things to all people has led a lot of developers to move their test suites over to it.

Most of the evidence to support the argument that Minitest is faster than RSpec consists of anecdotal blog posts and a few half-hearted benchmarks, but after running a few tests of my own, I found that there is a difference in execution time between equivalent suites using the two libraries which starts small and really starts to become significant as the number of tests increases above a thousand or so.

	100 tests	1K tests	10K tests
Minitest time	2.97s	24.04s	255.11s
RSpec time	3.10s	27.11s	466.67s
Ratio – RSpec / Minitest	104.32%	112.78%	182.93%

The choice of a testing framework is usually not without any kind of constraints. Fellow team members should have a say, and legacy concerns often make it difficult to change direction. Still, when these constraints are not present or can be resolved, selecting a smaller, nimbler testing framework is an easy decision that will pay off in the long term.

2. Load fewer gems with Bundler groups.

Most Rails applications I've worked on end up with a Gemfile that looks like a shopping list for nerds. For better or worse, the Ruby ecosystem makes it simple to assemble an infrastructure for your application from common components, thus freeing you to work on the essential logic. Bundler's Gemfile syntax lets you specify when a gem should only be loaded in a specific Rails environment.

```
gem "minitest", "~>5.4", group: :test
gem "minitest-rails", group: [:development, :test]
```

Bundler also supports a block syntax that lets you group environment-specific gems together:

```
group :development do
  gem "minitest-rails"
end

group :test do
  gem "minitest", "~>5.4"
  gem "minitest-rails"
end
```

Common Ruby gems that are environment-specific for either development or testing might include:

Development

- Spring
- Capistrano
- Brakeman
- Better Errors
- binding_of_caller (to enable Rails error page REPL)
- meta_request (for RailsPanel Chrome extension)

Testing

- Minitest
- Capybara
- factory_girl
- Faker, Forgery
- Database Cleaner
- Timecop

3. Use fixtures instead of factories.

For a number of years now, the prevailing wisdom has been that real Rails developers use factories to manage test data rather than the fixtures that ship with the framework. Some of this probably amounted to cargo-culting, but the major arguments against fixtures usually cast them as unmanageable or brittle as the application matured or that it obscured the intent of a test when the data was defined elsewhere.

But opinions are always in motion in the Rails world, and lately, a lot of developers have been coming back to fixtures. Part of this is probably due to changes in factory_girl that have made factory management feel a lot more like fixture management, but also, I suspect it's part of a larger movement toward simplifying application development and reducing the number of dependencies.

Regardless of how you feel about the relative merits of each method, one thing fixtures have going for them is speed.

- All your data is loaded into the test database via fast direct SQL calls before the first test runs, so your test instance comes preloaded with a complete, (hopefully) consistent set of test data that you can query and manipulate as needed.
- By minimizing the number of models created during test execution, you can also reduce the number of unintended operations and model creation resulting from ActiveRecord callbacks, Rails magic, etc.

4. Use an application preloader.

Major improvements in the Rails framework and the Ruby interpreter have made loading a barebones application environment much quicker than it used to be, but depending on the size of your application and the number and size of the gems you're using, you could still be looking at a delay of 10 seconds or more even before your first test is run.

To solve this problem, a number of developers have contributed *application preloaders* that speed up test and other command execution by keeping a version of your application running in a background process

Spring ships as part of the boilerplate Rails Gemfile and is maintained by the Rails core team. Whenever you make a request that needs a development or testing instance of your application (e.g. rake or rails), it forks a copy of the process running in the background. It's well integrated with the framework and easy to start using with any application using a recent version of Rails.

Zeus offers a different take on the same model as Spring, but in most respects, it works in very much the same way. Since it's not affiliated with Rails core as Spring is, the setup and integration is not quite as seamless.

Spring and Zeus make heavy use of `Process.fork` to work their magic, but that particular function isn't available in JRuby or on Windows. **Theine** was created to provide an application preloader for platforms that don't support forking.

5. Run your tests in parallel.

If you're like most Rails programmers, you're working on a fairly recent development rig with anywhere from 4-16 CPU cores. But when you run your suite, all test cases probably queue up and run in a single process on a single thread. That hardly seems fair.

One way to spread the load across more CPUs is by using Minitest's parallelization which allows you to distribute your tests across multiple workers just by adding a few lines to your **`test_helper.rb`** file:

```
require 'minitest/hell'

class Minitest::Test
  parallelize_me!
end
```

I wrote a blog post that goes into some detail about the ins and outs of using Minitest concurrent execution, but spoiler alert: the major disadvantage is that it uses Ruby's **`Thread`** class to implement the workers. That means that only Ruby implementations that support parallel execution (JRuby, Rubinius) will actually make use of multicore systems; others, most notably MRI, will not.

If you're using MRI, and you probably are, you should take a look at the parallel_tests gem which works by dividing your test case files into groups and spinning up multiple test processes that work completely independently, each using its own dedicated test database. While I prefer the Minitest method, parallel_tests gets results and provides a serviceable approach to the problem.

6. Stub external API and other HTTP calls.

Making calls to remote service APIs is common enough in modern web applications, and the Ruby ecosystem has spawned a long list of generic and specialized HTTP clients to help handle the task. When it comes to testing though, there are a number of reasons why we're better off skipping any remote service calls.

- Failure scenario testing – i.e. network errors, timeouts
- Error scenario testing – i.e. problems with authentication, access, request format, etc.
- Limited API access including rate limiting, etc.
- No control over service response times

This last point is particularly relevant to our discussion of performance. Even if our test suite only needs to make remote calls a dozen or so times, if the response time on each of those is one second... well, you can do the math yourself.

Fortunately, there's WebMock for just this sort of thing. After you add this gem to your bundle, you can disable all HTTP requests from your test environment by inserting a single line into your test_helper.rb:

```
WebMock.disable_net_connect!(:allow_localhost => true)
```

WebMock also gives you fine-grained control over specific HTTP requests and lets you stub responses for each so you can test your application's logic.

```
stub_request(:any, "www.example.com").
  to_return(:body => "abc", :status => 200, :headers => { 'Content-Length' => 3 })
```

7. Reduce unnecessary time-consuming operations.

Some developers will go out of their way to stub and mock any database operations in order to ensure that their tests are as fast and as isolated as they can be. While it might feel really good to have a complete test suite that runs in under 10 seconds, some developers punch giant holes in their test cases by zealously mocking and stubbing everything in sight. I tend to advocate a more moderate approach that plucks a lot of low-hanging fruit without unnecessarily deforming the natural relationships in the code.

- Increase your Rails log level by including one line in your **`test_helper.rb`**: `Rails.logger.level = Logger::FATAL`. It's an easy way to shave 5-10% off your suite execution time, and if you ever need additional insight into the logs, you can always change the log level temporarily to `Logger::DEBUG`.
- Don't use **`MyModel.create`** when **`MyModel.new`** will do. Calling create not only triggers a database insert operation, but it might also trigger the creation of other models as a result of callbacks being executed.
- Stub out model creation methods for controller tests. The logic in your controller actions should be simple enough that you'll only be interested in any operations performed on the model (e.g. create / save), which can be stubbed to return the desired response, and how the controller responds (e.g. response returned, template rendered, page redirected to, etc).
- Keep an eye out for other common operations that may take undue amounts of time such as:
  - Encryption and decryption, hashing functions
  - Image manipulation via ImageMagick or similar (ex: Paperclip thumbnail generation)

Ready to **level up** your testing?

I'm working on something you might like.

THE MINITEST COOKBOOK

The book will include step-by-step recipes and examples for users at any skill level from beginners to experienced pros designed to help you speed up your test suite and get more out of Minitest.

If that sounds like something you could use, click over to the website to sign up for the mailing list to receive updates and more freebies.

https://minitestcookbook.com