Extracted from:

Rails 5 Test Prescriptions

Build a Healthy Codebase

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rails 5 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

Edited by Katharine Dvorak

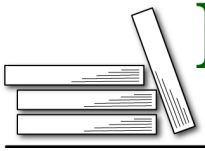
Rails 5 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Candace Cunningham
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-250-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

Testing Helper Methods

Helper modules are the storage attic of Rails applications. They are designed to contain reusable bits of view logic. This might include view-specific representations of data, or conditional logic that governs how content is displayed. Helper modules tend to get filled with all kinds of clutter that doesn't seem to belong anywhere else. Because they are a little tricky to set up for testing, helper methods often aren't tested even when they contain significant amounts of logic. Again, in my own practice I often move this logic into dedicated presenter classes, but you'll see a lot of code in Rails helpers you might want to unit-test.

RSpec helper tests go in spec/helpers. There's not a whole lot of special magic here—just a helper object that you use to call your helper methods.

Let's say you want to change your project view so behind-schedule projects show up differently. You could do that in a helper. My normal practice is to add a CSS class to the output for both the regular and behind-schedule cases, to give the design maximum freedom to display as desired.

Here's a test for that helper:

display/01/spec/helpers/projects_helper_spec.rb

```
Line 1  require "rails_helper"
-
-  RSpec.describe ProjectsHelper, type: :helper do
-    let(:project) { Project.new(name: "Project Runway") }
5
-    it "augments with status info when on schedule" do
-      allow(project).to receive(:on_schedule?).and_return(true)
-      actual = helper.name_with_status(project)
-      expect(actual).to have_selector("span.on_schedule", text: "Project Runway")
10   end
-  end
```

In this test you're creating a new project using a standard ActiveRecord new method. Rather than define a bunch of tasks that would mean the new project is on schedule, you just stub the on_schedule? method on line 7 to return true. This has the advantage of being faster than creating a bunch of objects and, I think, being more clear as to the exact state of the project being tested. You're using the have_selector matcher, as you did in Chapter 8, *Integration Testing with Capybara and Cucumber*, on page ?, to compare the expected HTML with the generated HTML.

That test will fail, however, because you haven't defined the name_with_status helper. Let's define one:

display/01/app/helpers/projects_helper.rb
```
module ProjectsHelper
  def name_with_status(project)
    content_tag(:span, project.name, class: "on_schedule")
  end
end
```

The test passes. Now let's add a second test for the remaining case. This test will look familiar:

display/02/spec/helpers/projects_helper_spec.rb
```
it "augments with status info when behind schedule" do
  allow(project).to receive(:on_schedule?).and_return(false)
  actual = helper.name_with_status(project)
  expect(actual).to have_selector(
    "span.behind_schedule", text: "Project Runway")
end
```

It passes with the following:

display/02/app/helpers/projects_helper.rb
```
module ProjectsHelper
  def name_with_status(project)
    dom_class = project.on_schedule? ? "on_schedule" : "behind_schedule"
    content_tag(:span, project.name, class: dom_class)
  end
end
```

One gotcha that you need to worry about when testing helpers is using Rails-internal view methods like url_for. Although all core Rails helpers are automatically loaded into the ActionView test environment, one or two have significant dependencies on the real controller object and therefore fail with opaque error messages during helper testing. The most notable of these is url_for. One workaround is to override url_for by defining it in your own test case. (The method signature is def url_for(options = {}).) The return value is up to you; a simple stub response is often good enough.

Sometimes helper methods take a block, which is expected to be ERB text (or the text of whatever template tool you're using to replace ERB). One common use of this kind of helper is access control, in which the logic in the helper determines whether the code in the block is invoked. Blocks also are very helpful as wrapper code for HTML that might surround many different kinds of text—a rounded-rectangle effect, for example.

Here's a simple example of a helper that takes a block:

```
def if_logged_in
  yield if logged_in?
end
```

It would be invoked like so:

```
<% if_logged_in do %>
 <%= link_to "logout", logout_path %>
<% end %>
```

To test the if_logged_in helper, you can take advantage of the fact that the yield statement is the last statement of the helper and, therefore, is the return value, and of the fact that Ruby will let you pass any arbitrary string into the block, giving you tests that look like this:

```
it "does not display if not logged_in" do
  expect(logged_in?).to be_falsy
  expect(if_logged_in { "logged in" }).to be_nil
end
```

```
it "displays if logged in" do
  login_as users(:quentin)
  expect(logged_in?).to be_truthy
  expect(if_logged_in { "logged in" }).to eq("logged in")
end
```

The first test asserts that the block is not invoked, so the helper returns nil. The second asserts that the block is invoked, just returning the value passed into the block.

You have to be a little careful here because these tests are just testing the helper method's return value, not what is sent to the output stream. The output-stream part is a side effect of the process, but it is stored in a variable called output_buffer, which you can access via testing. So you could write the preceding tests as follows:

```
it "does not display if not logged_in" do
  expect(logged_in?).to be_falsy
  if_logged_in { "logged in" }
  expect(output_buffer).to be_nil
end

it "displays if logged in" do
  login_as users(:quentin)
  expect(logged_in?).to be_truthy
  if_logged_in { "logged in" }
  expect(output_buffer).to eq("logged in")
end
```

If for some reason your helper method requires a specific instance variable to be set, cut that out immediately; it's a bad idea. However, if you must use an instance variable in your helper and you want to test it in RSpec, use the assigns method, as in expect(assigns(:project)).to eq(Project.find(1)). That tests that there is an instance variable named @project that has been set to a specific project.

Testing Controllers and Requests

The biggest change in Rails 5 testing when compared to earlier versions is the deprecation of some controller-testing functionality. In this section we'll first look at the RSpec features in Rails 5, then we'll go back and look at how controller testing worked in previous versions of Rails.

For most of this book I've focused on just what Rails 5 has to offer, and not spent time comparing Rails 5 to previous versions. I'm going to make a partial exception here and discuss the deprecated controller features as part of this section, for two reasons:

- Experienced Rails coders new to Rails 5 likely use these features and it's worth talking about what fills the same ecological niche in Rails 5.
- Even if you are a new Rails coder, you are likely to encounter older Rails code that uses controller tests, and you probably want to see what is going on.

RSpec for Rails 5 has two test types that are designed to test a single controller action. They are very similar. *Request specs* are basically wrappers around Rails integration tests, and *controller tests* are basically RSpec wrappers around Rails 5 controller tests. (Rails has always used the name "feature tests" for what RSpec calls "controller tests," which is confusing, because RSpec uses "feature tests" to indicate Capybara tests.) The older controller test behavior, should you need it to pull up a legacy codebase, can be reinstated with the rails-controller-testing gem.[1]

The naming gets confusing, but the basic taxonomy of when to use each test is not that bad. Here are the guidelines for RSpec:

- Use a request spec if you're testing the results of a single request to the Rails server, there's no interaction with the user in the test, and you're checking the behavior of the controller action or a side effect of calling the controller action.
- Use an RSpec system spec with Capybara, as described in Chapter 8, *Integration Testing with Capybara and Cucumber*, on page ?, if you're testing something that requires user interaction, such as simulated clicks or multiple page interactions, and if you're testing against the final result on the page.
- Use a system spec with the :js metadata if the integration test requires JavaScript.

If you're used to older versions of RSpec, be advised that I'm explicitly recommending using request specs in place of controller specs, and using system specs in place of feature specs (for reasons described in Chapter 8, *Integration Testing with Capybara and Cucumber*, on page ?).

Minitest and the standard Rails tools have a slightly different decision pattern, which I cover in Chapter 12, *Minitest*, on page ?.

1. https://github.com/rails/rails-controller-testing

Simulating Requests

Ideally, your controllers are relatively simple. The complicated functionality is in a model or other object and is being tested in your unit tests for those objects. One reason this is a best practice is that models are easier to test than requests because they're easier to extract and use independently in a test framework.

Prescription 25	A request test should test behavior of a single request. A request test should not fail because of problems in the model.

A request test that overlaps with model behavior is part of the awkward middle ground of testing that the Rails 5 changes are designed to avoid. If the request test is going to the database, then the test is slower than it needs to be. And if a model failure can cascade into the controller tests, then it's harder than it needs to be to isolate the problem.

A request test should have one or more of the following goals:

- Verify that a normal, basic user request triggers expected model or workflow object calls. Test doubles work extremely well for this purpose.
- Verify that a side effect, such as an email or background job, is triggered as part of a user request.
- Verify that an ill-formed or otherwise invalid user request is handled properly, for whatever definition of "properly" fits your app.
- Verify security, such as requiring logins for pages as needed and testing that users who enter a URL for a resource they shouldn't be able to see are blocked or diverted. (I discuss this more in Chapter 13, *Testing for Security*, on page ?.) This is also a good candidate to be offloaded into unit testing with, for example, the Pundit gem.[2]

Your request tests in Rails 5 will usually start with a simulated request. To make this simulation easier, Rails provides a test method to simulate each HTTP verb: `delete`, `get`, `head`, `patch`, `post`, and `put`. Each of these methods works the same way. (Internally, they all dispatch to a common method that does all the work.) A full call to one of these methods has one argument and up to five optional keyword arguments. The argument is a URL string, which can be passed as a string or as a Rails URL helper. The keyword arguments are, alphabetically:

2. https://github.com/elabs/pundit

- as: Specifies the content type you want returned, especially if that content type is :json.
- env: A hash of environment variables. I admit it's not immediately clear to me why you'd want to set those in a request spec.
- headers: A hash of simulated header information you might want passed.
- params: The params hash for controllers, including anything that might be in a query string or a route variable. You can use the params hash to cover items that might otherwise be arguments to the URL.
- xhr: If true, tells the controller to treat the fake request as an Ajax call.

So, a contrived call might look like this:

```
get(projects_url, params: {id: @project.id}, xhr: true, as: :json)
```

If one of the param arguments is an uploaded file—say, from a multipart form—you can simulate that using the Rails helper fixture_file_upload(filename, mime_type), like this:

```
post(:create,
  params: {logo: fixture_file_upload('/test/data/logo.png', 'image/png')}
```

Third-party tools to manage uploads, such as Paperclip and CarrierWave, typically have more specific testing helpers.

If you've done Rails controller tests before, you'll notice that this is an API change. Most notably, you can no longer set the session or the flash via request tests. Rails now considers those to be internal implementation details.