Extracted from:

Rails 4 Test Prescriptions

Build a Healthy Codebase

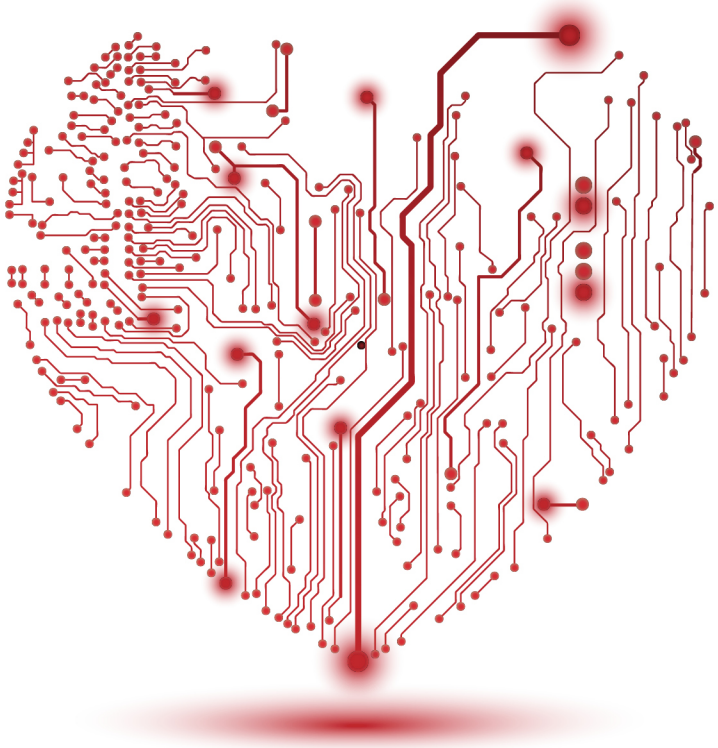
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Rails 4 Test Prescriptions

Build a Healthy Codebase



Noel Rappin

Edited by Lynn Beighley

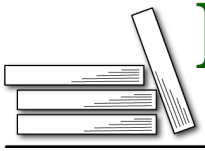
Rails 4 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

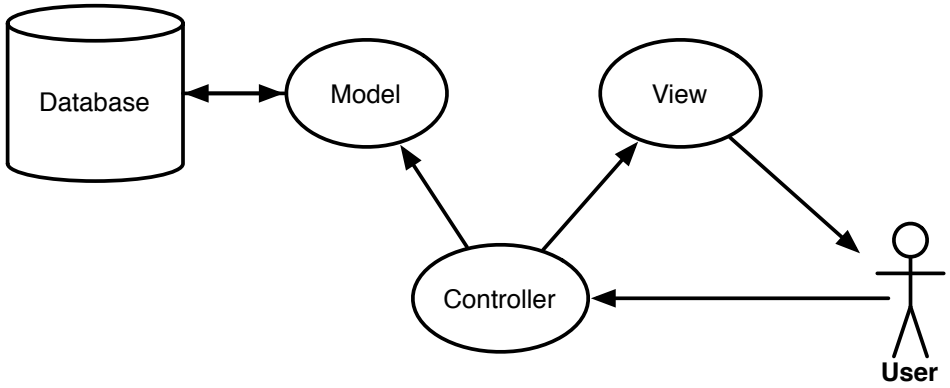
The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

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

Printed in the United States of America.
ISBN-13: 978-1-941222-19-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2014

Rails applications follow a model-view-controller, or MVC, pattern. The view layer has the responsibility of presenting data to the user, which in a server-side web application usually means generating HTML. Ideally, the view layer does this with minimal interaction with the model. The controller takes in information about the user request, contacts the appropriate parts of the model layer for data, and passes that information on to the view layer. The following is a very simplified diagram.



Testing Rails controllers and views is more challenging than testing Rails models. You can see from the diagram that controllers and views both interact with the external users, whereas models are more inherently isolated. In Rails, controller and view instances are typically created by the framework itself and are not easy to create in isolation during a test. (As far as the Rails developer is concerned, the view instance is mostly just a template.) Controller and view calls often are more interesting for their side effects than for the value they return. Also, individual controller actions and view templates are often too large to be meaningfully unit-tested.

While the Rails framework and third-party testing tools allow us to interact with controller actions and view templates in our test environment, the issues of isolation and size still exist. A discussion of how to best test views and controllers, therefore, often turns into a discussion about what code belongs in the controller and view and what should be extracted into a different object. The object extracted to, which is not a part of the Rails framework, is sometimes referred to as a PORO: Plain Old Ruby Object. The issue of how to best deconstruct or refactor controller and view code is somewhat contentious within the Rails community.

We've discussed the idea that the most useful tests test either an entire end-to-end process or a single unit. Controller and view tests are easy to put in the middle ground and are therefore notoriously brittle and hard to manage.

Testing Controllers

We've already written a few controller tests as part of our earlier testing walkthroughs. Let's take a look at one of them:

display/01/gatherer/spec/controllers/projects_controller_spec.rb

```
Line 1  require 'rails_helper'
2
3  RSpec.describe ProjectsController, type: :controller do
4
5    describe "POST create" do
6      it "creates a project" do
7        post :create, project: {name: "Runway", tasks: "Start something:2"}
8        expect(response).to redirect_to(projects_path)
9        expect(assigns(:action).project.name).to eq("Runway")
10     end
```

This test is simple but has most of the features of a basic controller test. Like many tests we have seen, controller tests have three parts. First, the controller test may create data needed to cover a particular logic path. We don't need any data for this test, but we will see examples of generating controller-specific test data in our next examples. Second, on line 7 the code performs an action. Specifically, it simulates a post request to the controller's create action with one argument, the hash {name: "Runway", tasks: "start something:2"}, which represents the parameters being passed to the action as part of the request.

Finally, on lines 8 and 9, our test makes assertions about the controller's behavior. Broadly, we care about two kinds of behavior. We care about what template or other action the controller passes control to. The redirect_to matcher is one of a few assertions added by RSpec in controller test groups to specify that transfer of control. We may also care that the controller specifies particular instance variables for use by a view template. The assigns method is also managed by RSpec controller groups to enable assertions to be made about those values.

What to Test in a Controller Test

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 controllers because they are generally easier to extract and use independently in a test framework.

> **Prescription 20** A controller test should test controller behavior. A controller test should not fail because of problems in the model.

A controller test that overlaps with model behavior is part of the awkward middle ground of testing that we're trying to avoid. If the controller test is actually 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 controller test should have one or more of the following goals:

- Verifying that a normal, basic user request triggers expected model calls and passes the necessary data to the view.
- Verifying that an ill-formed or otherwise invalid user request is handled properly, for whatever definition of "properly" fits your app.
- Verifying 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. We will discuss this more in Chapter 11, *Testing for Security*, on page ?.

Simulating Requests in a Controller Test

Most of your controller tests in Rails will surround a simulated request. To make this simulation easier, Rails provides a controller test method for 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 five arguments, though you'll often just use the first three:

```
get :show, {id: @task.id}, {user_id: "3",
    current_project: @project.id.to_s}, {notice: "flash test"}
```

The method name, `get`, is the HTTP verb being simulated—sort of. While the controller test will set the HTTP verb if for some reason you query the Rails request object, it does not check the Rails routing table to see if that action is reachable using that HTTP verb. As a result, you can't test routing via a controller test. Rails does provide a mechanism for testing routes, which we'll cover in *Testing Routes*, on page 11.

The first argument—in this case `:show`—is the controller action being called. The second argument, `{id: @task.id}`, is a hash that becomes the `params` hash in the controller action. In the controller action called from this test, you would expect `params[:id]` to equal `@task.id`. The Rails form name-parsing trick is not

used here—if you want to simulate a form upload, you use a multilevel hash directly, as in user: {name: "Noel", email: "noel@noelrappin.com"}, which implies params[:user][:name] == "Noel" in the controller.

Any value passed in this hash argument is converted to a string—specifically, to_param is called on it. So you can do something like id: 3, confident that it will be "3" in the controller. This, by the way, is a welcome change in recent versions of Rails; older versions did not do this conversion, which led to occasional heads pounding against walls.

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

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

If you're using a third-party tool, such as Paperclip or CarrierWave to manage uploads, those tools typically have more specific testing helpers.

The fourth and fifth arguments to these controller methods are optional and rarely used. The fourth argument sets key/value pairs for the session object, which is useful in testing multistep processes that use the session for continuity. The fifth argument represents the Rails flash object, which is useful...well, never, but if for some reason the incoming flash is important for your logic, there it is.

You may occasionally want to do something fancier to the simulated request. In a controller test you have access to the request object as @request, and access to the controller object as @controller. (As you'll see in *Evaluating Controller Results, on page 9*, you also have the @response object.) You can get at the HTTP headers using the hash @request.headers.

There is one more controller action method, xml_http_request (also aliased to xhr). This simulates a classic Ajax call to the controller and has a slightly different signature:

```
it "makes an ajax call" do
  xhr :post, :create, :task => {:id => "3"}
end
```

The method name is xhr, the first argument is the HTTP verb associated with the xhr call, and the remaining arguments are the arguments to all the other controller-calling methods in the same order: action, params, session, and flash. The xhr call sets the appropriate headers such that the Rails controller will appropriately be able to consider the request an Ajax request (meaning .js

format blocks will be triggered), then simulates the call based on its arguments.

Evaluating Controller Results

A controller test has three things you might want to validate after the controller action:

- Did it return the expected HTTP status code? RSpec provides the response.status object and the have_http_status matcher for this purpose.
- Did it pass control to the expected template or redirected controller action? Here we have the render_template and redirect_to matchers.
- Did it set the values that the view will expect? For this we have the special hash objects assigns, cookies, flash, and session.

Often you'll combine more than one of these in the same test:

```
it "is a successful index request with no filters" do
  get :index
  expect(response).to have_http_status(:success)
  expect(response).to render_template(:index)
end
```

Asserting Controller Response Type

Let's talk about these three types of assertions in more detail:

We can use have_http_status to verify the HTTP response code sent back to the browser from Rails. Normally we use this assertion to ensure that our controller correctly distinguishes between success and redirect or error cases.

The value passed to have_http_status is usually one of four special symbols:

Symbol	HTTP Code Equivalent	Symbol	HTTP Code Equivalent
:success	200–299	:redirect	300–399
:missing	404	:error	500–599

If you need to test for a more specific response, you can pass either the exact HTTP code number or one of the associated symbols defined by Rack.[1] Note that RSpec uses the codes defined by SYMBOL_TO_STATUS_CODE. The most common case I've had for specific HTTP codes is the need to distinguish between 301 permanent redirects (:moved-permanently) and other redirects.

Asserting Which View Is Rendered

The render_template matcher is used to determine whether the controller is passing control to the expected view template. The method has a simple form and then some optional complexity. In the simple form, render_template is passed a template name that is specified exactly as it would be in the controller, using render :action—the template name can be a string or a symbol. If the argument is just a single string or symbol, then it is checked against the name of the main template that rendered the action.

Normally I will not employ render_template when the controller action is just using the implicit Rails default and ceding to a view of the same name. I will use render_template when I expect the controller will need to explicitly pass control to a specific template, with the most common case being a create action that is successful and renders a show template.

When you expect the controller to redirect, you can use redirect_to to assert the exact nature of the redirect. The argument to redirect_to is pretty much anything Rails can convert to a URL, although the method's behavior is slightly different based on what the argument actually is. The code for redirect_to explicitly includes have_http_status(:redirect), so you don't need to duplicate that assertion.

If the argument to redirect_to is the name of a URL because it's a string or a method for a Rails named route, or because it's an object that has a Rails RESTful route, then the assertion passes if and only if the redirecting URL exactly matches the asserted URL. For testing purposes, Rails will assume that the application's local hostname is http://www.example.com. If that's too exact a test for your taste, you can pass a hash to redirect_to, which specifies the :controller, :action, and any parameters. If the argument is a hash, then

1. https://github.com/rack/rack/blob/master/lib/rack/utils.rb

assert_redirected_to checks only the exact key/value pairs in the hash; other parts of the URL are not checked.

Rails controller tests do not—repeat, *do not*—follow the redirect. Any data-validation tests you write apply only to the method before the redirect occurs. If you need your test to follow the redirection for some reason, you are cordially invited to try something in an integration test; see Chapter 10, *Integration Testing with Capybara and Cucumber*, on page ?.

Asserting Controller Data

Rails allows you to verify the data generated by the controller action being tested with the four items mentioned earlier: assigns, session, cookies, and flash. Of these, assigns, which gives access to instance variables declared by the controller, is the most commonly used. A typical use might look like this, with a common use of assigns and an admittedly contrived use of session:

```
it "shows a task" do
  task = Task.create!
  get :show, id: task.id
  expect(response).to have_http_status(:success)
  expect(assigns(:task).id).to eq(task.id)
  expect(session[:previous_page]).to eq("task/show")
end
```

The cookies and flash special variables are used similarly, though I don't write tests for the flash very often. The cookie hash has key/value pairs only for cookies. If you want to test other cookie attributes, you need to access them via the request object.

Testing Routes

Although the basics of Rails routing are simple, the desire to customize Rails' response to URLs can lead to confusion about exactly what your application is going to do when converting between a URL and a Rails action. Rails provides a way to specify route behavior in a test.

Routing tests are not typically part of my TDD process—usually my integration test implicitly covers the routing. That said, sometimes routing gets complicated and has some logic of its own (especially if you're trying to replicate an existing URL scheme), so it's nice to have this as part of your test suite.

RSpec-Rails puts route tests in the spec/routing directory. The primary matcher that RSpec-Rails uses for route testing is route_to. Here's a sample test that includes all seven default RESTful routes for the project resource:

display/01/gatherer/spec/routing/project_routing_spec.rb
```
require 'rails_helper'

RSpec.describe "project routing" do
  it "routes projects" do
    expect(get: "/projects").to route_to(
        controller: "projects", action: "index")
    expect(post: "/projects").to route_to(
        controller: "projects", action: "create")
    expect(get: "/projects/new").to route_to(
        controller: "projects", action: "new")
    expect(get: "/projects/1").to route_to(
        controller: "projects", action: "show", id: "1")
    expect(get: "/projects/1/edit").to route_to(
        controller: "projects", action: "edit", id: "1")
    expect(patch: "/projects/1").to route_to(
        controller: "projects", action: "update", id: "1")
    expect(delete: "/projects/1").to route_to(
        controller: "projects", action: "destroy", id: "1")
  end
end
```

All of these are using the same form. The argument to expect is a key/value pair where the key is the HTTP verb and the value is the string form of the route. The argument to route_to is a set of key/value pairs where the keys are the parts of the calculated route (including controller, action, and what have you) and the values are, well, the values.

The route_to matcher tests the routes in both directions. It checks that when you send the path through the routing engine, you get the controller, action, and other variables specified. It also checks that the set of controller, action, and other variables sent through the router results in the path string (which is why you might need to specify query-string elements). It's not clear to me why a route might pass in one direction and not the other.

RSpec also provides a be_routable method, which is designed to be used in the negative to show that a specific path—say, the Rails default—is not recognized:

```
expect(get: "/projects/search/fred").not_to be_routable
```

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.

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 we want to change our project view so behind-schedule projects show up differently. We 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/gatherer/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" 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 we'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, we 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.

We're using the have_selector matcher again to compare the expected HTML with the generated HTML. We'll cover have_selector in more detail when we talk about Capybara.

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

display/01/gatherer/app/helpers/projects_helper.rb

```
Line 1  module ProjectsHelper
2
3    def name_with_status(project)
4      content_tag(:span, project.name, class: 'on_schedule')
5    end
6  end
```

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

display/02/gatherer/spec/helpers/projects_helper_spec.rb

```
it "augments project name 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/gatherer/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 view-testing 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. 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
```

Which would be invoked like so:

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

To test the if_logged_in helper, we 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 us pass any arbitrary string into the block, giving us 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
```