6 Ways to Remove Pain From Feature Testing in Ruby on Rails

Mitch Lloyd posted on October 2, 2013

Lire cet article en français.

Writing feature tests in Ruby on Rails used to be the most painful part of my development work flow. Now I kind of like it. Here's what's different:

1. Don't Use Cucumber

Disclaimer: The views expressed in the following paragraph do not necessarily represent those of the Staff or Partners of Gaslight Software, LLC.

If you have Cucumber installed, uninstall it. This stuff is already hard enough without trying to parse natural language into Ruby code.

I'm using:

- Rspec - Testing DSL
- FactoryGirl - Model Builder
- Capybara - DOM Dominator
- Database Cleaner - Database Cleaner
- Spring - Startup Speeder

I'm happy with these. Let's make a spec.

```
feature 'Navigating through workpapers' do
  let(:user) { create(:user) }
  let(:audit) { create(:audit, users: [user]) }

  scenario "User sees workpapers within an audit" do
    workpaper = create(:workpaper, audit: audit)

    visit '/'
    fill_in 'email', with: user.email
    fill_in 'password', with: 'password'
    click_on 'Log In'

    find('#audit-selector').select audit.name
    expect(page).to have_css?('.workpaper', text: workpaper.name)
  end
end
```

This doesn't look too bad but a more complicated feature would really start to get muddy and the login logic will eventually be duplicated between tests. Even this example doesn't read as well as I would like.

2. Use Page Objects

Capybara selectors are likely to break as development continues. Your copy-writer decides the login button should say "Sign into a world of possibilities". Now you need to fix your tests.

Page objects are adapters to the specifics of your DOM. When the markup changes, you'll know exactly where to go to fix the issues.

Here's a login page object:

```
class LoginPage
  include Capybara::DSL

  def visit_page
    visit '/'
    self
  end

  def login(user)
    fill_in 'email', with: user.email
    fill_in 'password', with: 'password'
    click_on 'Log In'
  end
end
```

Here is another page object for the Workpaper index page:

```
class WorkpaperIndexPage
  include Capybara::DSL

  def select_audit(audit)
    find('#audit-selector').select audit.name
  end

  def has_workpaper?(workpaper)
    has_css?('.workpaper', text: workpaper.name)
  end
end
```

Here's the new test using these page objects:

```
feature 'Navigating through workpapers' do
  let(:user) { create(:user) }
  let(:audit) { create(:audit, users: [user]) }
  let(:login_page) { LoginPage.new }
  let(:workpaper_page) { WorkpaperIndexPage.new }

  scenario "User sees workpapers within an audit" do
    workpaper = create(:workpaper, audit: audit)

    login_page.visit_page.login(user)
    workpaper_page.select(audit)
    expect(workpaper_page).to have_workpaper(workpaper)
  end
end
```

Now let's say someone keeps tweaking that login button. You can pop into the Login page, start using an ID selector or an I18n entry (which might even have been a good idea upfront), and not worry about changing any other test code. All of that page fiddlyness is contained in the page objects.

These page objects are very minimal, but they can grow to provide additional error checking as the user navigates through the UI or pass off other pages (or sections of pages) to assert on and interact with. Page objects pay off so often and so much that **I always use page objects in my feature tests**. Just as I never write SQL in my Rails views, I never access the DOM in a feature test without a page object.

3. Create Useful Failure Messages

Failing feature tests can be hard to diagnose. Let's say you used a page object like this:

```
expect(workpaper_page).to have_one_workpaper(workpaper)

Failure/Error: expect(workpaper_page).to have_one_workpaper(workpaper)
  expected #has_one_workpaper?(workpaper) to return true, got false
```

The expectation is readable, but it would be even better to know whether the workpaper was missing or whether there were other workpapers there causing the expectation to fail.

In practice I've raised exceptions when calling these type of predicate methods on page objects.

```
Failure/Error: expect(workpaper_page).to have_one_workpaper(workpaper)
  PageExpectationNotMetError:
    expected one workpaper called "My Sweet Workpaper", but the following
    workpapers were on the page:
      * "Bogus Workpaper"
      * "My Sweet Workpaper"
```

I've been using this technique sparingly and I'm still looking for a more elegant approach, but this helps me to get more descriptive error messages and avoid debugging trips to the browser. Let me know if you have some good

ways of integrating messages like this into your tests.

4. Embrace Asynchronous Testing

Many frustrations with browser automation testing stem from mistakes using assertions that need to wait. Adding a sleep to your tests an okay way to debug code if you think you have a timing issue, but `sleep` should should never make it into your final test code.

Flickering tests (tests that fail intermittently) will kill your confidence in your test suit. They should be fixed or deleted.

In general, my best advice is to learn the Capybara API well. Here are some pointers:

- Using `#all` does not wait, so this is probably not the matcher you want.
- The `#has_css?` method takes a `count` parameter so that you can indicate how many matching elements you want to wait for.
- Asserting something like `expect(page).to_not have_css('.post')` is usually not what you want. This matcher waits for `.post` elements to show up before it can pass, resulting in a big delay. Usually you'll want to use `expect(page).to have_no_css('.post')` which will pass immediately if the elements are not there, but wait for them to disappear if they are there. To use this matcher with confidence you'll first want to assert that there were posts at some point earlier.

At times you may want to wait for something to happen outside of Capybara. For that, this handy `eventually` helper will save the day:

The following code waits for the workpaper to be awesome and fails if the expectation is not met within 2 seconds.

```
eventually { expect(workpaper).to be_awesome }
```

But when would you ever make a polling assertion like this without Capybara? Read on...

5. Get Serious About Data Building

Early on I remember hearing a mantra for feature tests that went something like "Do everything from the perspective of the user". This advice primarily served to discourage testers from manually manipulating your data in feature tests. I can confidently say that this was bad advice. It's impractical to sign up a new user with a credit card and complete 20 other provisioning steps just to click an approval button.

I use FactoryGirl extensively for setting up data in tests. This means that I have factories that that can generate complicated objects. For instance here is a way to make a `workpaper` with a `worflow` that has `steps` assigned to certain `users` called `preparers` and `reviewers`.

```
FactoryGirl.define do
  factory :workpaper do
    sequence(:name) {|n| "workpaper #{n}"}

    factory :assigned_workpaper do
      ignore do
        preparer { create(:user) }
        reviewer { create(:user) }
      end

      after(:create) do |workpaper, evaluator|
        create(:assigned_workflow, workpaper: workpaper, preparer: evaluator.preparer, reviewer: evaluator.reviewer)
      end
    end
  end

  factory :workflow do
    factory :assigned_workflow do
      ignore do
        preparer { create(:user) }
        reviewer false
      end

      after(:create) do |workflow, evaluator|
        create(:step, workflow: workflow, user: evaluator.preparer)

        if evaluator.reviewer
          create(:step, workflow: workflow, user: evaluator.reviewer)
        end
      end
    end
  end

  factory :step
end
```

This lets me create specific, declarative objects for my tests:

```
create(:assigned_workpaper, preparer: first_user, reviewer: second_user)
```

I always create database models through FactoryGirl in my feature tests. I'm definitely a fan of FactoryGirl, but I think there is probably room for improvement for building complex data like this. Whatever tool you use, the data setup for your tests should be readable, easy to use, and well-factored.

Not only is it acceptable to setup data before you begin your test, it's also acceptable to assert that another side effect occurred that is not immediately visible to users. For instance, in the world of rich client web apps, seeing something on the screen doesn't mean that everything has been successfully persisted in the database.

Just as we have helpers for building up the data, we should have helpers for inspecting it. This assertion will make sure that the preparer for a workpaper has been successfully saved in the database:

```
eventually { preparer_for(workpaper).should be(preparer) }
```

6. Prefer Refining Existing Tests Over Creating New Ones

When I started doing feature tests in Rails I got some advice that went something like "each test should have one user action and one assertion". So I worked like this:

1. Write a cucumber scenario for a feature.
2. Make the code work.
3. Write another cucumber scenario for a new aspect of that feature.
4. Make the code work.

While this is a useful guideline for unit tests, this is not good advice for feature tests.

Let's say I have a test like this:

```
scenario "assigning a reviewer to a workpaper" do
  user_visits_workpaper(user, workpaper)
  ui.begin_assigning_reviewer
  ui.assign_work_to(reviewer)
  eventually { expect(reviewer_for workpaper).to eq(other_tester) }
end
```

When we call `ui.begin_assigning_reviewer` a modal dialog pops up that lets the user pick someone to assign as a reviewer. That feature works. Great.

Now I want to make sure that only users with access to review show up in that modal dialog. Instead of making a new spec for that, I would rather refine the one that I already have.

```
scenario "assigning a reviewer to a workpaper" do
  user_visits_workpaper(user, workpaper)
  ui.begin_assigning_reviewer
  expect(ui).to have_excluded_user(non_reviewer)
  ui.assign_work_to(reviewer)
  eventually { expect(reviewer_for workpaper).to eq(other_tester) }
end
```

You would not want to use this approach in unit tests, but this works in feature tests where the goal is to guide your progress and efficiently detect regressions.

But Who's Testing Your Tests?

When you start building a lot of logic into you tests someone will eventually say "But who's testing your tests?" to imply that your tests are too complicated or over-engineered. Your production code tests your tests. Don't use the sentiment in this question to justify poorly-factored, unreadable feature tests.

The specific tools and techniques discussed above will change over time but I have heightened my sensitivity to bad feature tests forever. Refactor aggressively, design thoughtfully, and love your feature tests.

Share
reddit this!
« Previous Post
Next Post »

Get the Gaslight Newsletter
Our brightest ideas delivered monthly

Email Address Email Address
Subscribe

- Work
- Services
- Process
- About
- Blog
- Training

- Contact
- FAQs
- Careers
- Podcast
- Coffee Friday

Connect with us

-
-
-

-

Made with love in Cincinnati

• hello@teamgaslight.com