Team

Capybara

09.01.2017

Nguyen Thuan
Neolab Vietnam Co., Ltd
60 Nguyen Dinh Chieu
Ho Chi Minh City, District 1

Overview

Capybara helps you test web applications by simulating how a real user would interact with your app. It is agnostic about the driver running your tests and comes with Rack::Test and Selenium support built in. WebKit is supported through an external gem.

Install

1. **Gemfile**: gem 'capybara'
2. **Make directory** in spec/features (for Capybara 2)

Reference:
http://stackoverflow.com/questions/14180003/rspec-naming-conventions-for-files-and-directory-structure

Additional

Automation Test

1. **Gemfile**
   a. gem "capybara-webkit"
   b. gem "selenium-webdriver"
   c. gem "capybara-screenshot"
2. **Dependence packages:** sudo apt-get install libqt4-dev libqtwebkit-dev
3. **Install Geckodriver**

Download at https://github.com/mozilla/geckodriver/releases/

```
tar -xvzf geckodriver-*

chmod +x geckodriver
cp geckodriver /usr/local/bin/
```

Reference:
https://github.com/thoughtbot/capybara-webkit/wiki/Installing-Qt-and-compiling-capybara-webkit#ubuntu-trusty-1404

Usage

I. Feature Specs vs. Views Specs

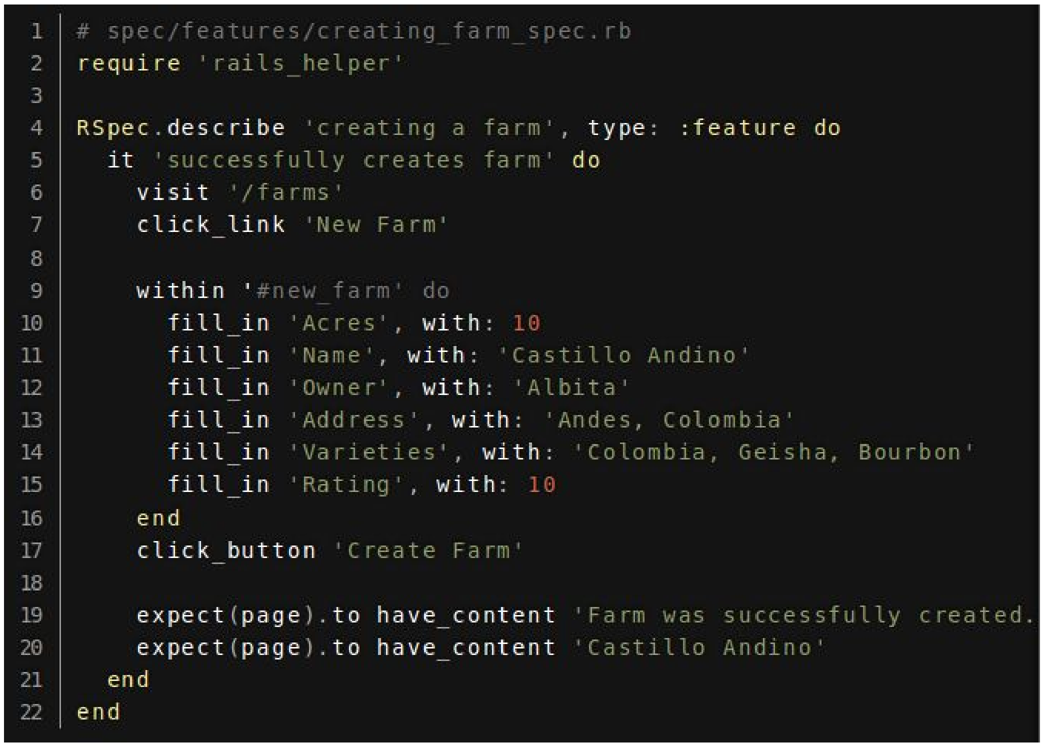
Reference:
http://stackoverflow.com/questions/35258592/is-there-any-difference-between-a-feature-spec-and-a-view-spec

A **feature spec** uses a headless browser to test the entire system from the outside just like a user uses it. It exercises code, database, views and Javascript too, if you use the right headless browser and turn on Javascript.

```ruby
feature 'User creates a foobar' do
  scenario 'they see the foobar on the page' do
    visit new_foobar_path

    fill_in 'Name', with: 'My foobar'
    click_button 'Create Foobar'

    expect(page).to have_css '.foobar-name', 'My foobar'
  end
end
```



```ruby
# spec/features/creating_farm_spec.rb
require 'rails_helper'

RSpec.describe 'creating a farm', type: :feature do
  it 'successfully creates farm' do
    visit '/farms'
    click_link 'New Farm'

    within '#new_farm' do
      fill_in 'Acres', with: 10
      fill_in 'Name', with: 'Castillo Andino'
      fill_in 'Owner', with: 'Albita'
      fill_in 'Address', with: 'Andes, Colombia'
      fill_in 'Varieties', with: 'Colombia, Geisha, Bourbon'
      fill_in 'Rating', with: 10
    end
    click_button 'Create Farm'

    expect(page).to have_content 'Farm was successfully created.
    expect(page).to have_content 'Castillo Andino'
  end
end
```

A **view spec** just renders a view in isolation, with template variables provided by the test rather than by controllers.

```
describe 'products/_product.html.erb' do
  context 'when the product has a url' do
    it 'displays the url' do
      assign(:product, build(:product, url: 'http://example.com')

      render

      expect(rendered).to have_link 'Product', href: 'http://example.cc
    end
  end

  context 'when the product url is nil' do
    it "displays 'None'" do
      assign(:product, build(:product, url: nil)

      render

      expect(rendered).to have_content 'None'
    end
  end
end
```

II. References

1. Navigating

```
visit('/entries')
visit(entry_comments_path(entry))
```

The visit method only takes a single parameter, the request method is always **GET**.

You can get the current path of the browsing session, and test it using the **have_current_path** matcher:

```
expect(page).to have_current_path(post_comments_path(post))
```

2. Clicking links and buttons

You can interact with the webapp by following links and buttons. Capybara automatically follows any redirects, and submits forms associated with buttons.

```
click_link('id-of-link')
click_link('Link Text')
click_button('Save')
click_on('Link Text') # clicks on either links or buttons
click_on('Button Value')
```

Reference:
http://www.rubydoc.info/github/teamcapybara/capybara/master/Capybara/Node/Actions

3. Interacting with forms

There are a number of tools for interacting with form elements:

```
fill_in('First Name', with: 'John')
fill_in('Password', with: 'Seekrit')
fill_in('Description', with: 'Really Long Text...')
choose('A Radio Button')
check('A Checkbox')
uncheck('A Checkbox')
attach_file('Image', '/path/to/image.jpg')
select('Option', from: 'Select Box')
```

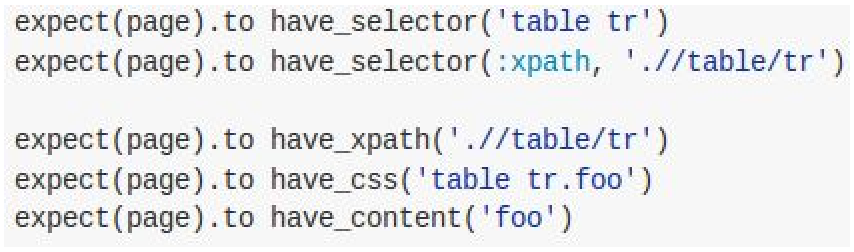
Reference:
http://www.rubydoc.info/github/teamcapybara/capybara/master/Capybara/Node/Actions

4. Querying - Matcher

```
page.has_selector?('table tr')
page.has_selector?(:xpath, './/table/tr')

page.has_xpath?('.//table/tr')
page.has_css?('table tr.foo')
page.has_content?('foo')
```

Rspec matchers



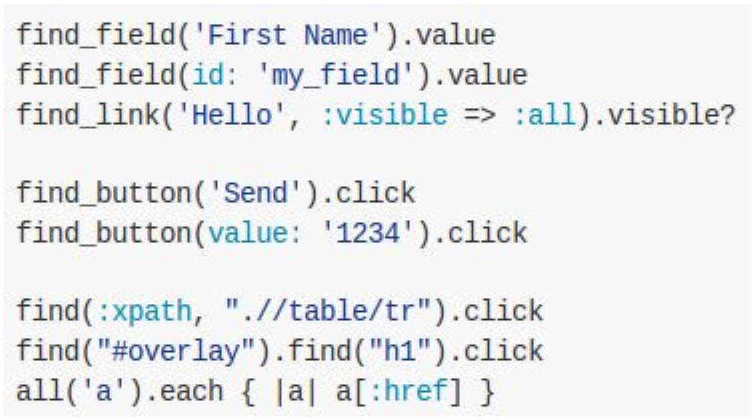
```
expect(page).to have_selector('table tr')
expect(page).to have_selector(:xpath, './/table/tr')

expect(page).to have_xpath('.//table/tr')
expect(page).to have_css('table tr.foo')
expect(page).to have_content('foo')
```

Reference:
http://www.rubydoc.info/github/teamcapybara/capybara/master/Capybara/Node/Matchers

5. Finding

You can also find specific elements, in order to manipulate them:

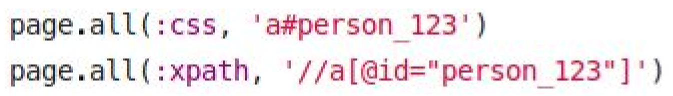


```
find_field('First Name').value
find_field(id: 'my_field').value
find_link('Hello', :visible => :all).visible?

find_button('Send').click
find_button(value: '1234').click

find(:xpath, ".//table/tr").click
find("#overlay").find("h1").click
all('a').each { |a| a[:href] }
```

Example



```
page.all(:css, 'a#person_123')
page.all(:xpath, '//a[@id="person_123"]')
```

# Find all elements on the page matching the given selector and options.
# If the type of selector is left out, Capybara uses Capybara.default_selector. It's set to :css
# by default.

You can custom selector by "**add_selector**" and "**modify_selector"**

Add selector:
http://www.rubydoc.info/github/teamcapybara/capybara/Capybara#add_selector-class_method

```
Capybara.add_selector(:row) do
  xpath { |num| ".//tbody/tr[#{num}]" }
end
```

This makes it possible to use this selector in a variety of ways:

```
find(:row, 3)
page.find('table#myTable').find(:row, 3).text
page.find('table#myTable').has_selector?(:row, 3)
within(:row, 3) { expect(page).to have_content('$100.000') }
```

Modify selector:

http://www.rubydoc.info/github/teamcapybara/capybara/Capybara#modify_selector-class_method

6. Scoping

Capybara makes it possible to restrict certain actions, such as interacting with forms or clicking links and buttons, to within a specific area of the page. For this purpose you can use the generic within method. Optionally you can specify which kind of selector to use.

```
within("li#employee") do
  fill_in 'Name', with: 'Jimmy'
end

within(:xpath, ".//li[@id='employee']") do
  fill_in 'Name', with: 'Jimmy'
end
```

There are special methods for restricting the scope to a specific fieldset, identified by either an id or the text of the fieldset's legend tag, and to a specific table, identified by either id or text of the table's caption tag.

```
within_fieldset('Employee') do
  fill_in 'Name', with: 'Jimmy'
end

within_table('Employee') do
  fill_in 'Name', with: 'Jimmy'
end
```

Example

```
within('table') do
end

within('#profiles-table', text: "Should be a caption") do
end

within_table('profiles-table') do
end

within_table(text: 'Should be a caption') do
end
```

7. Working with windows

Capybara provides some methods to ease finding and switching windows:

```
facebook_window = window_opened_by do
  click_button 'Like'
end
within_window facebook_window do
  find('#login_email').set('a@example.com')
  find('#login_password').set('qwerty')
  click_button 'Submit'
end
```

Example

```
within_window "Log In | Facebook" do
  fill_in 'Email:', with: fb_user.email
  fill_in 'Password:', with: fb_user.password
  click_button "Log In"
  # syncronization makes this never return, maybe because
  # it's running in a different window?
  without_resyncronize { click_button "Allow" }
end
wait_a_while_for { page.should have_content(content) }
```

8. Scripting

In drivers which support it, you can easily execute JavaScript:

```
page.execute_script("$('body').empty()")
```

For simple expressions, you can return the result of the script. Note that this may break with more complicated expressions:

```
result = page.evaluate_script('4 + 4');
```

Example

session.execute_script '$("table tbody tr:first-child").css("background-color", "green")'

9. Modals

In drivers which support it, you can accept, dismiss and respond to alerts, confirms and prompts.

You can accept or dismiss alert messages by wrapping the code that produces an alert in a block:

```
accept_alert do
  click_link('Show Alert')
end
```

You can accept or dismiss a confirmation by wrapping it in a block, as well:

```
dismiss_confirm do
  click_link('Show Confirm')
end
```

You can accept or dismiss prompts as well, and also provide text to fill in for the response:

```
accept_prompt(with: 'Linus Torvalds') do
  click_link('Show Prompt About Linux')
end
```

All modal methods return the message that was presented. So, you can access the prompt message by assigning the return to a variable:

```
message = accept_prompt(with: 'Linus Torvalds') do
  click_link('Show Prompt About Linux')
end
expect(message).to eq('Who is the chief architect of Linux?')
```

Example:

session.driver.browser.switch_to.alert.accept

10. Debugging

It can be useful to take a snapshot of the page as it currently is and take a look at it:

- *save_and_open_page*
- *print page.html*
- *page.save_screenshot('screenshot.png')*

- *save_and_open_screenshot*

Screenshots are saved to Capybara.save_path, relative to the app directory. If you have required capybara/rails, Capybara.save_path will default to tmp/capybara.

Capybara to the Rescue

Much of Capybara's source code is dedicated to battling asynchronous problem.

Capybara.default_max_wait_time = 5 (default: 2)

1. Find the first matching element

- **Bad:** first(".active").click

If there isn't an *.active* element on the page yet, first will return *nil* and the click will fail.

- **Good:**

```
# If you want to make sure there's exactly one
find(".active").click

# If you just want the first element
find(".active", match: :first).click
```

Capybara will wait for the element to appear before trying to click.

2. Interact with all matching elements

- **Bad:** all(".active").each(&:click)

If there are no matching elements yet, an empty array will be returned, and no elements will be affected.

- **Good:**

```
find(".active", match: :first)
all(".active").each(&:click)
```

Capybara will wait for the first matching element before trying to click on the rest.

3. Directly interacting with JavaScript

- **Bad:** execute_script("$('.active').focus()")

JavaScript expressions may be evaluated before the action is complete, and the wrong element or no element may be affected.

- **Good:**

```
# If you want to make sure there's exactly one
find(".active")

# If you just want the first element
execute_script("$('.active').focus()")
```

Capybara will wait until a matching element is on the page, and then dispatch a JavaScript command which interacts with it.

4. **Checking a field's value**

- **Bad:** expect(find_field("Username").value).to eq("Joe")

Capybara will wait for the matching element and then immediately return its value. If the value changes from a page load or Ajax request, it will be too late.

- **Good:** expect(page).to have_field("Username", with: "Joe")

Capybara will wait for a matching element and then wait until its value matches, up to two seconds.

5. **Checking an element's attribute**

- **Bad:** expect(find(".user")["data-name"]).to eq("Joe")

Capybara will wait for the matching element and then immediately return the requested attribute.

- **Good:** expect(page).to have_css(".user[data-name='Joe']")

Capybara will wait for the element to appear and have the correct attribute.