Table of Contents (left)

Capybara

- github.com/jnicklas/capybara

Description:

Capybara aims to simplify the process of integration testing Rack applications, such as Rails, Sinatra or Merb. Capybara simulates how a real user would interact with a web application. It is agnostic about the driver running your tests and currently comes with Rack::Test and Selenium support built in. HtmlUnit, WebKit and env.js are supported through external gems.

A complete reference is available at at rubydoc.info.

Install:

Install as a gem:

```
sudo gem install capybara
```

On OSX you may have to install libffi, you can install it via MacPorts with:

```
sudo port install libffi
```

Development:

- Source hosted at GitHub.
- Please direct questions, discussion or problems to the mailing list.
- If you found a reproducible bug, open a GitHub Issue to submit a bug report.

Pull requests are very welcome (and even better than bug reports)! Make sure your patches are well tested, Capybara is a testing tool after all. Please create a topic branch for every separate change you make.

Capybara uses bundler in development. To set up a development environment, simply do:

```
git submodule update --init
gem install bundler
bundle install
```

Using Capybara with Cucumber

Capybara is built to work nicely with Cucumber. Support for Capybara is built into cucumber-rails. In your Rails app, just run:

```
rails generate cucumber:install --capybara
```

And everything should be set up and ready to go.

If you want to use Capybara with Cucumber outside Rails (for example with Merb or Sinatra), you'll need to require Capybara and set the Rack app manually:

```
require 'capybara/cucumber'
Capybara.app = MyRackApp
```

Now you can use it in your steps:

```
When /I sign in/ do
  within("#session") do
    fill_in 'Login', :with => 'user@example.com'
    fill_in 'Password', :with => 'password'
  end
  click_link 'Sign in'
end
```

Capybara sets up some tags for you to use in Cucumber. Often you'll want to run only some scenarios with a driver that supports JavaScript, Capybara makes this easy: simply tag the scenario (or feature) with `@javascript`:

```
@javascript
Scenario: do something Ajaxy
  When I click the Ajax link
  ...
```

You can change which driver Capybara uses for JavaScript:

```
Capybara.javascript_driver = :culerity
```

There are also explicit `@selenium`, `@culerity` and `@rack_test` tags set up for you.

Using Capybara with RSpec

If you prefer RSpec to using Cucumber, you can use the built in RSpec support by adding the following line (typically to your `spec_helper.rb` file):

```
require 'capybara/rspec'
```

You can now write your specs like so:

```
describe "the signup process", :type => :request do
  before :each do
    User.make(:email => 'user@example.com', :password => 'caplin')
  end

  it "signs me in" do
    within("#session") do
      fill_in 'Login', :with => 'user@example.com'
      fill_in 'Password', :with => 'password'
    end
    click_link 'Sign in'
  end
end
```

Capybara is only included in example groups tagged with `:type => :request` (or `:acceptance` for compatibility with Steak).

If you are testing a Rails app and using the `rspec-rails` gem, these `:request` example groups may look familiar to you. That's because they are RSpec versions of Rails integration tests. So, in this case essentially what you are getting are Capybara-enhanced request specs. This means that you can use the Capybara helpers *and* you have access to things like named route helpers in your tests (so you are able to say, for instance, `visit edit_user_path(user)`, instead of `visit "/users/#{user.id}/edit"`, if you prefer that sort of thing). A good place to put these specs is `spec/requests`, as `rspec-rails` will automatically tag them with `:type => :request`. (In fact, `spec/integration` and `spec/acceptance` will work just as well.)

`rspec-rails` will also automatically include Capybara in `:controller` and `:mailer` example groups.

RSpec's metadata feature can be used to switch to a different driver. Use `:js => true` to switch to the javascript driver, or provide a `:driver` option to switch to one specific driver. For example:

```
describe 'some stuff which requires js', :js => true do
  it 'will use the default js driver'
  it 'will switch to one specific driver', :driver => :celerity
end
```

Finally, Capybara also comes with a built in DSL for creating descriptive acceptance tests:

```
feature "Signing up" do
  background do
    User.make(:email => 'user@example.com', :password => 'caplin')
  end

  scenario "Signing in with correct credentials" do
    within("#session") do
      fill_in 'Login', :with => 'user@example.com'
      fill_in 'Password', :with => 'caplin'
    end
    click_link 'Sign in'
  end
end
```

This is, in fact, just a shortcut for making a request spec, where `feature` is an alias for `describe ..., :type => :request`, `background` is an alias for `before`, and `scenario` is an alias for `it/specify`.

Note that Capybara's built in RSpec support only works with RSpec 2.0 or later. You'll need to roll your own for earlier versions of RSpec.

Using Capybara with Test::Unit

To use Capybara with Test::Unit, include the DSL (`include Capybara` up until version 0.4.x, `include Capybara::DSL` for newer versions) in whatever test class you are using. For example, if your classes derive from `ActionDispatch::IntegrationTest`, use

```
class ActionDispatch::IntegrationTest
  include Capybara::DSL
end
```

Test::Unit does not support selecting the driver through test metadata, but you can switch the driver for specific classes using the `setup` and `teardown` methods. See the section "Selecting the

Driver”.

Using Capybara with Ruby on Rails

If you are using the Rails framework, add this line to automatically configure Capybara to test against your Rails application:

```
require 'capybara/rails'
```

Using Capybara with Rack

If you’re using Capybara with a non-Rails Rack application, set `Capybara.app` to your application class:

```
Capybara.app = MyRackApp
```

Drivers

Capybara uses the same DSL to drive a variety of browser and headless drivers.

Selecting the Driver

By default, Capybara uses the `:rack_test` driver, which is fast but does not support JavaScript. You can set up a different default driver for your features. For example if you’d prefer to run everything in Selenium, you could do:

```
Capybara.default_driver = :selenium
```

However, if you are using RSpec or Cucumber, you may instead want to consider leaving the faster `:rack_test` as the `default_driver`, and marking only those tests that require a JavaScript-capable driver using `:js => true` or `@javascript`, respectively. By default, JavaScript tests are run using the `:selenium` driver. You can change this by setting `Capybara.javascript_driver`.

You can also change the driver temporarily (typically in the Before/setup and After/teardown blocks):

```
Capybara.current_driver = :culerity  # temporarily select different driver
... tests ...
Capybara.use_default_driver  # switch back to default driver
```

Note that switching the driver creates a new session, so you may not be able to switch in the middle of a test.

RackTest

RackTest is Capybara’s default driver. It is written in pure Ruby and does not have any support for executing JavaScript. Since the RackTest driver works directly agains the Rack interface, it does not need any server to be started, it can work directly work against any Rack app. This means that if your application is not a Rack application (Rails, Sinatra and most other Ruby frameworks are Rack applications) then you cannot use this driver. You cannot use the RackTest driver to test a remote application. capybara-mechanize intends to provide a similar driver which works against remote servers, it is a separate project.

RackTest can be configured with a set of headers like this:

```
Capybara.register_driver :rack_test do |app|
  Capybara::RackTest::Driver.new(app, :browser => :chrome)
end
```

See the section on adding and configuring drivers.

Selenium

At the moment, Capybara supports Selenium 2.0 (Webdriver), **not** Selenium RC. Provided Firefox is installed, everything is set up for you, and you should be able to start using Selenium right away.

Capybara can block and wait for Ajax requests to finish after you've interacted with the page. To enable this behaviour, set the `:resynchronize` driver option to `true`. This should normally not be necessary, since Capybara's automatic reloading should take care of any asynchronicity problems. See the section on Asynchronous JavaScript for details.

Note: Selenium does not support transactional fixtures; see the section "Transactional Fixtures" below.

HtmlUnit

There are three different drivers, maintained as external gems, that you can use to drive HtmlUnit:

- Akephalos might be the best HtmlUnit driver right now.
- Celerity only runs on JRuby, so you'll need to install the celerity gem under JRuby: `jruby -S gem install celerity`
- Culerity: Install celerity as noted above, and make sure that JRuby is in your path. Note that Culerity does not seem to be working under Ruby 1.9 at the moment.

Note: HtmlUnit does not support transactional fixtures; see the section "Transactional Fixtures" below.

env.js

The capybara-envjs driver uses the envjs gem (GitHub, rubygems.org) to interpret JavaScript outside the browser. The driver is installed by installing the capybara-envjs gem:

```
gem install capybara-envjs
```

More info about the driver and env.js are available through the links above. The envjs gem only supports Ruby 1.8.7 at this time.

Note: Envjs does not support transactional fixtures; see the section "Transactional Fixtures" below.

Capybara-webkit

The capybara-webkit drive is for true headless testing. It uses WebKitQt to start a rendering engine process. It can execute JavaScript as well. It is significantly faster than drivers like Selenium since it does not load an entire browser.

You can install it with:

```
gem install capybara-webkit
```

And you can use it by:

```
Capybara.javascript_driver = :webkit
```

The DSL

Capybara's DSL (domain-specific language) is inspired by Webrat. While backwards compatibility is retained in a lot of cases, there are certain important differences. Unlike in Webrat, all searches in Capybara are *case sensitive*. This is because Capybara heavily uses XPath, which doesn't support case insensitivity.

Navigating

You can use the `visit` method to navigate to other pages:

```
visit('/projects')
visit(post_comments_path(post))
```

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

You can get the current path of the browsing session for test assertions:

```
current_path.should == post_comments_path(post)
```

Clicking links and buttons

Full reference: Capybara::Node::Actions

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')
```

Interacting with forms

Full reference: Capybara::Node::Actions

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')
```

Querying

Full reference: Capybara::Node::Matchers

Capybara has a rich set of options for querying the page for the existence of certain elements, and working with and manipulating those elements.

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

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

You can use these with RSpec's magic matchers:

```
page.should have_selector('table tr')
page.should have_selector(:xpath, '//table/tr')
page.should have_no_selector(:content)

page.should have_xpath('//table/tr')
page.should have_css('table tr.foo')
page.should have_content('foo')
page.should have_no_content('foo')
```

Note that `page.should have_no_xpath` is preferred over `page.should_not have_xpath`. Read the section on asynchronous JavaScript for an explanation.

If all else fails, you can also use the `page.html` method to test against the raw HTML:

```
page.html.should match /<span>.../i
```

Finding

Full reference: Capybara::Node::Finders

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

```
find_field('First Name').value
find_link('Hello').visible?
find_button('Send').click

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

Note that `find` will wait for an element to appear on the page, as explained in the Ajax section. If the element does not appear it will raise an error.

These elements all have all the Capybara DSL methods available, so you can restrict them to specific parts of the page:

```
find('#navigation').click_link('Home')
find('#navigation').should have_button('Sign out')
```

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
```

Note that `within` will scope the actions to the *first* (not *any*) element that matches the selector.

There are special methods for restricting the scope to a specific fieldset, identified by either an id or the text of the fieldet'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
```

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');
```

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
```

Transactional fixtures

Transactional fixtures only work in the default Rack::Test driver, but not for other drivers like

Selenium. Cucumber takes care of this automatically, but with Test::Unit or RSpec, you may have to use the database_cleaner gem. See this explanation (and code for solution 2 and solution 3) for details.

Asynchronous JavaScript (Ajax and friends)

When working with asynchronous JavaScript, you might come across situations where you are attempting to interact with an element which is not yet present on the page. Capybara automatically deals with this by waiting for elements to appear on the page.

When issuing instructions to the DSL such as:

```
click_link('foo')
click_link('bar')
page.should have_content('baz')
```

If clicking on the **foo** link triggers an asynchronous process, such as an Ajax request, which, when complete will add the **bar** link to the page, clicking on the **bar** link would be expected to fail, since that link doesn't exist yet. However Capybara is smart enought to retry finding the link for a brief period of time before giving up and throwing an error. The same is true of the next line, which looks for the content **baz** on the page; it will retry looking for that content for a brief time. You can adjust how long this period is (the default is 2 seconds):

```
Capybara.default_wait_time = 5
```

Be aware that because of this behaviour, the following two statements are **not** equivalent, and you should **always** use the latter!

```
page.should_not have_xpath('a')
page.should have_no_xpath('a')
```

The former would incorrectly wait for the content to appear, since the asynchronous process has not yet removed the element from the page, it would therefore fail, even though the code might be working correctly. The latter correctly waits for the element to disappear from the page.

Capybara's waiting behaviour is quite advanced, and can deal with situations such as the following line of code:

```
find('#sidebar').find('h1').should have_content('Something')
```

Even if JavaScript causes `#sidebar` to disappear off the page, Capybara will automatically reload it and any elements it contains. So if an AJAX request causes the contents of `#sidebar` to change, which would update the text of the `h1` to "Something", and this happened, this test would pass. If you do not want this behaviour, you can set `Capybara.automatic_reload` to `false`.

Using the DSL in unsupported testing frameworks

You can mix the DSL into any context by including `Capybara::DSL`:

```
require 'capybara'
require 'capybara/dsl'

Capybara.default_driver = :culerity
```

```
module MyModule
  include Capybara::DSL

  def login!
    within("//form[@id='session']") do
      fill_in 'Login', :with => 'user@example.com'
      fill_in 'Password', :with => 'password'
    end
    click_link 'Sign in'
  end
end
```

Calling remote servers

Normally Capybara expects to be testing an in-process Rack application, but you can also use it to talk to a web server running anywhere on the internets, by setting app_host:

```
Capybara.current_driver = :selenium
Capybara.app_host = 'http://www.google.com'
...
visit('/')
```

Note that the default driver (`:rack_test`) does not support running against a remote server. With drivers that support it, you can also visit any URL directly:

```
visit('http://www.google.com')
```

By default Capybara will try to boot a rack application automatically. You might want to switch off Capybara's rack server if you are running against a remote application:

```
Capybara.run_server = false
```

Using the sessions manually

For ultimate control, you can instantiate and use a Session manually.

```
require 'capybara'

session = Capybara::Session.new(:culerity, my_rack_app)
session.within("//form[@id='session']") do
  session.fill_in 'Login', :with => 'user@example.com'
  session.fill_in 'Password', :with => 'password'
end
session.click_link 'Sign in'
```

XPath, CSS and selectors

Capybara does not try to guess what kind of selector you are going to give it, and will always use CSS by default. If you want to use XPath, you'll need to do:

```
within(:xpath, '//ul/li') { ... }
find(:xpath, '//ul/li').text
find(:xpath, '//li[contains(.//a[@href = "#"]/text(), "foo")]').value
```

Alternatively you can set the default selector to XPath:

```
Capybara.default_selector = :xpath
find('//ul/li').text
```

Capybara allows you to add custom selectors, which can be very useful if you find yourself using the same kinds of selectors very often:

```
Capybara.add_selector(:id) do
  xpath { |id| XPath.descendant[XPath.attr(:id) == id.to_s] }
end

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

The block given to xpath must always return an XPath expression as a String, or an XPath expression generated through the XPath gem. You can now use these selectors like this:

```
find(:id, 'post_123')
find(:row, 3)
```

You can specify an optional match option which will automatically use the selector if it matches the argument:

```
Capybara.add_selector(:id) do
  xpath { |id| XPath.descendant[XPath.attr(:id) == id.to_s] }
  match { |value| value.is_a?(Symbol) }
end
```

Now use it like this:

```
find(:post_123)
```

This :id selector is already built into Capybara by default, so you don't need to add it yourself.

Beware the XPath // trap

In XPath the expression // means something very specific, and it might not be what you think. Contrary to common belief, // means "anywhere in the document" not "anywhere in the current context". As an example:

```
page.find(:xpath, '//body').all(:xpath, '//script')
```

You might expect this to find all script tags in the body, but actually, it finds all script tags in the entire document, not only those in the body! What you're looking for is the .// expression which means "any descendant of the current node":

```
page.find(:xpath, '//body').all(:xpath, './/script')
```

The same thing goes for within:

```
within(:xpath, '//body') do
  page.find(:xpath, './/script')
  within(:xpath, './/table/tbody') do
    ...
  end
end
```

Configuring and adding drivers

Capybara makes it convenient to switch between different drivers. It also exposes an API to tweak those drivers with whatever settings you want, or to add your own drivers. This is how to switch the selenium driver to use chrome:

```
Capybara.register_driver :selenium do |app|
  Capybara::Selenium::Driver.new(app, :browser => :chrome)
end
```

However, it's also possible to give this a different name, so tests can switch between using different browsers effortlessly:

```
Capybara.register_driver :selenium_chrome do |app|
  Capybara::Selenium::Driver.new(app, :browser => :chrome)
end
```

Whatever is returned from the block should conform to the API described by Capybara::Driver::Base, it does not however have to inherit from this class. Gems can use this API to add their own drivers to Capybara.

The Selenium wiki has additional info about how the underlying driver can be configured.

Gotchas:

- Access to session and request is not possible from the test, Access to response is limited. Some drivers allow access to response headers and HTTP status code, but this kind of functionality is not provided by some drivers, such as Selenium.
- Access to Rails specific stuff (such as `controller`) is unavailable, since we're not using Rails' integration testing.
- Freezing time: It's common practice to mock out the Time so that features that depend on the current Date work as expected. This can be problematic, since Capybara's Ajax timing uses the system time, resulting in Capybara never timing out and just hanging when a failure occurs. It's still possible to use plugins which allow you to travel in time, rather than freeze time. One such plugin is Timecop.
- When using Rack::Test, beware if attempting to visit absolute URLs. For example, a session might not be shared between visits to `posts_path` and `posts_url`. If testing an absolute URL in an Action Mailer email, set `default_url_options` to match the Rails default of `www.example.com`.

License:

Generated on Tue Nov 15 12:13:48 2011 by yard 0.7.3 (ruby-1.9.2).