github github

Advanced Search

Search…

Search

- Explore
- Gist
- Blog
- Help

 pmq20

- Notifications
- Account Settings
- Log Out

jnicklas / capybara

- Watch Unwatch
- Fork
- 2,163
- 350

- Code
- Network
- Pull Requests 2
- Issues 15
- Wiki 2
- Stats & Graphs

Acceptance test framework for web applications — Read more

- ZIP

- HTTP
- Git Read-Only

https://github.com/jnicklas

Read-Only access

- Current branch: master
  Switch Branches/Tags
  Filter branches/tags
  - Branches
  - Tags

  0.4.1.1_stable

  1.0_stable

  1.1_stable

advanced_matchers

bundler

drivers

extendable_selectors

master

rack_test_refactring

xpath

- Files
- Commits
- Branches 10
- Tags 26
- Downloads 0

Latest commit to the **master** branch

Add more referer tests …

```
This includes a pending test "should send no referer when visiting a
second page", which does not pass on Rack::Test right now.
```

commit 7b5682e272

 joliss authored about 12 hours ago

capybara /

name	age	history message
features	May 11, 2011	Fixed typo [joealba]
lib	about 12 hours ago	Add more referer tests [joliss]
spec	about 13 hours ago	Moved test to shared_example_for in driver, included test both for ra… [Rob van Dijk]
.gitignore	August 30, 2011	ignore chromedriver log [Jonas Nicklas and Nicklas Ramhöj]
.gitmodules	January 09, 2011	Add xpath gem as submodule and run from submodule [jnicklas]
.rspec	November 17, 2010	Added fuubar [jnicklas]
.travis.yml	November 06, 2011	added travis build config [jnicklas]
Gemfile	January 09, 2011	Add xpath gem as submodule and run from submodule [jnicklas]
History.txt	about 12 hours ago	Update History [joliss]
License.txt	January 03, 2012	Move license to License.txt and update copyright span [joliss]
README.md	about 12 hours ago	Typo. Thanks @khustochka! [joliss]
Rakefile	4 days ago	Update gemspec and Rakefile for README.md and License.txt [joliss]
capybara.gemspec	4 days ago	Update gemspec and Rakefile for README.md and License.txt [joliss]
xpath- 375ec5c	January 03, 2012	Upgrade XPath [jnicklas]

README.md

Capybara

Capybara helps you test Rails and Rack 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.

Need help? Ask on the mailing list (please do not open an issue on GitHub): http://groups.google.com/group/ruby-capybara

Setup

To install, type

```
sudo gem install capybara
```

If you are using Rails, add this line to your test helper file:

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

If you are not using Rails, set Capybara.app to your rack app:

```
Capybara.app = MyRackApp
```

Using Capybara with Cucumber

The `cucumber-rails` gem comes with Capybara support built-in. If you are not using Rails, manually load the `capybara/cucumber` module:

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

You can use the Capybara DSL in your steps, like so:

```
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
```

You can switch to the `Capybara.javascript_driver` (`:selenium` by default) by tagging scenarios (or features) with `@javascript`:

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

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

Using Capybara with RSpec

Load RSpec 2.x support by adding the following line (typically to your `spec_helper.rb` file):

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

If you are using Rails, put your Capybara specs in `spec/requests` or `spec/integration`.

If you are not using Rails, tag all the example groups in which you want to use Capybara with `:type => :request`.

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
```

Use `:js => true` to switch to the `Capybara.javascript_driver` (`:selenium` by default), 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 => :webkit
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
```

`feature` is in fact just an alias for `describe ..., :type => :request`, `background` is an alias for `before`, and `scenario` for `it`.

Using Capybara with Test::Unit

- If you are using Rails, add `database_cleaner` to your Gemfile:

  ```
  group :test do
    gem 'database_cleaner'
  end
  ```

  Then add the following code in your `test_helper.rb` file to make Capybara available in all test cases deriving from `ActionDispatch::IntegrationTest`:

  ```
  # Transactional fixtures do not work with Selenium tests, because Capybara
  # uses a separate server thread, which the transactions would be hidden
  # from. We hence use DatabaseCleaner to truncate our test database.
  DatabaseCleaner.strategy = :truncation

  class ActionDispatch::IntegrationTest
    # Make the Capybara DSL available in all integration tests
    include Capybara::DSL

    # Stop ActiveRecord from wrapping tests in transactions
    self.use_transactional_fixtures = false

    teardown do
      DatabaseCleaner.clean       # Truncate the database
      Capybara.reset_sessions!    # Forget the (simulated) browser state
      Capybara.use_default_driver # Revert Capybara.current_driver to Capybara.default_driver
    end
  end
  ```

- If you are not using Rails, define a base class for your Capybara tests like so:

```
class CapybaraTestCase < Test::Unit::TestCase
  include Capybara::DSL

  def teardown
    Capybara.reset_sessions!
    Capybara.use_default_driver
  end
end
```

  Remember to call `super` in any subclasses that override `teardown`.

To switch the driver, set `Capybara.current_driver`. For instance,

```
class BlogTest < ActionDispatch::IntegrationTest
  setup do
    Capybara.current_driver = Capybara.javascript_driver # :selenium by default
  end

  test 'shows blog posts'
    # ... this test is run with Selenium ...
  end
end
```

Using Capybara with MiniTest::Spec

Set up your base class as with Test::Unit. (On Rails, the right base class could be something other than ActionDispatch::IntegrationTest.)

The capybara_minitest_spec gem (Github, rubygems.org) provides MiniTest::Spec expectations for Capybara. For example:

```
page.must_have_content('Important!')
```

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

Note: 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 against the Rack interface, it does not need any server to be started, it can work directly 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: drivers which run the server in a different thread may not work share the same transaction as your tests, causing data not to be shared between your test and test server, see "Transactions and database setup" below.

Capybara-webkit

The capybara-webkit driver is for true headless testing. It uses QtWebKit 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

A complete reference is available at at rubydoc.info.

Note: 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')
```

Note: The negative forms like `has_no_selector?` are different from `not has_selector?`. Read the section on asynchronous JavaScript for an explanation.

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')
```

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: `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: `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
```

You can also retrieve the current state of the DOM as a string using `page.html`.

```
print page.html
```

This is mostly useful for debugging. You should avoid testing against the contents of `page.html` and use the more expressive finder methods instead.

Transactions and database setup

Some Capybara drivers need to run against an actual HTTP server. Capybara takes care of this and starts one for you in the same process as your test, but on another thread. Selenium is one of those drivers, whereas RackTest is

not.

If you are using an SQL database, it is common to run every test in a transaction, which is rolled back at the end of the test, rspec-rails does this by default out of the box for example. Since transactions are usually not shared across threads, this will cause data you have put into the database in your test code to be invisible to Capybara.

Cucumber handles this by using truncation instead of transactions, i.e. they empty out the entire database after each test. You can get the same behaviour by using a gem such as database_cleaner.

It is also possible to force your ORM to use the same transaction for all threads. This may have thread safety implications and could cause strange failures, so use caution with this approach. It can be implemented in ActiveRecord through the following monkey patch:

```
class ActiveRecord::Base
  mattr_accessor :shared_connection
  @@shared_connection = nil

  def self.connection
    @@shared_connection || retrieve_connection
  end
end
ActiveRecord::Base.shared_connection = ActiveRecord::Base.connection
```

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.has_xpath?('a')
page.has_no_xpath?('a')
```

The former would immediately fail because the content has not yet been removed. Only the latter would wait for the asynchronous process to remove the content from the page.

Capybara's Rspec matchers, however, are smart enough to handle either form. The two following statements are functionally equivalent:

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

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 = :webkit

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: 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(:webkit, 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

Capybara.add_selector(:flash_type) do
  css { |type| "#flash.#{type}" }
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)
find(:flash_type, :notice)
```

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`.

Development

If you found a *reproducible* bug, open a GitHub Issue to submit a bug report.

Even better, send a pull request! Make sure all changes are well tested, Capybara is a testing tool after all. Topic branches are good.

To set up a development environment, simply do:

```
git submodule update --init
gem install bundler
bundle install
bundle exec rake  # run the test suite
```

- Training
- GitHub Enterprise
- Site Status

Tools

- Gauges: Analyze web traffic
- Speaker Deck: Presentations
- Gist: Code snippets
- GitHub for Mac
- Issues for iPhone
- Job Board

Extras

- GitHub Shop
- The Octodex

Documentation

- GitHub Help
- Developer API
- GitHub Flavored Markdown
- GitHub Pages

- Terms of Service
- Privacy
- Security


Powered by the Dedicated Servers and Cloud Computing of Rackspace Hosting®

Markdown Cheat Sheet

Format Text

Headers

```
# This is an <h1> tag
## This is an <h2> tag
###### This is an <h6> tag
```

Text styles

```
*This text will be italic*
_This will also be italic_
**This text will be bold**
__This will also be bold__

*You **can** combine them*
```

Lists

Unordered

```
* Item 1
* Item 2
```

```
  * Item 2a
  * Item 2b
```

Ordered

```
1. Item 1
2. Item 2
3. Item 3
   * Item 3a
   * Item 3b
```

Miscellaneous

Images

```
![GitHub Logo](/images/logo.png)
Format: ![Alt Text](url)
```

Links

```
http://github.com - automatic!
[GitHub](http://github.com)
```

Blockquotes

```
As Kanye West said:

> We're living the future so
> the present is our past.
```

Code Examples in Markdown

Syntax highlighting with GFM

```
```javascript
function fancyAlert(arg) {
  if(arg) {
    $.facebox({div:'#foo'})
  }
}
```
```

Or, indent your code 4 spaces

```
Here is a Python code example
without syntax highlighting:

    def foo:
      if not bar:
        return true
```

Inline code for comments

```
I think you should use an
`<addr>` element here instead.
```

Something went wrong with that request. Please try again. Dismiss