Feature Testing at Pega using WebDriver, Ruby, RSpec & Capybara

To do:

Add some links to tutorials and references on RSpec & Capybara, TDD & BDD.

Contents

Introduction

The combination of the WebDriver API with Ruby, RSpec and Capybara lets you write powerful automated tests for web applications. You can also test using only a subset of these tools, but they provide more power and flexibility when used together.

Webdriver – is a protocol (or API) for controlling web browsers from other programs.

Ruby – is a popular scripting language

RSpec – is a framework for test driven development

Capybara – is a library for automated browser testing

The remainder of this document will show you how to create and run automated tests at Pega using these tools.

- First we will discuss using them in your own development environment, with browsers on Windows and Macintosh.
- Then we will add mobile browsers into the mix.
- Later we will discuss how to have your tests run as part of Pega's automated test environment.
- Finally we will discuss best practices.

Getting Started on your computer **with WebDriver, Ruby, RSpec & Capybara**

Note to Mac users – you will need administrative privileges to execute many of the commands below. Your account must have those privileges by default, or you must be able to use the **`sudo`** *command.*

Ruby

Which version of Ruby to download? We recommend version 1.9.3 for Windows, and 2.1.1 for Mac.

Download and install Ruby from here:
https://www.ruby-lang.org/en/downloads/
Note that you may be directed to another site: rubyinstaller.org.

When installing Ruby, you may encounter some prompts:

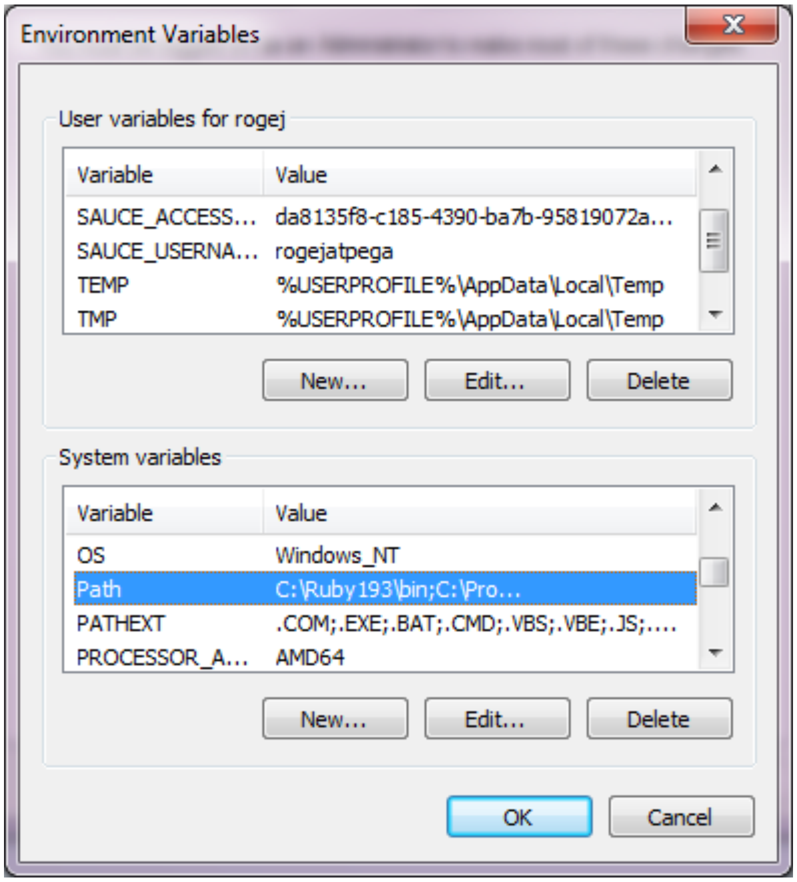
- Tcl/Tk support – this is not needed.
- Add Ruby executables to path – this is recommended.

- Associate .rb and .rbw files – this is recommended.

Ensure that the ruby/bin directory is in your Path. For example, a typical installation on Windows will result in a directory C:\Ruby193 and so you may need to add the following to your Path environment variable:

```
C:\Ruby193\bin
```

See the screen shot below:



Environment Variables

User variables for rogej

Variable	Value
SAUCE_ACCESS...	da8135f8-c185-4390-ba7b-95819072a...
SAUCE_USERNA...	rogejatpega
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
OS	Windows_NT
Path	C:\Ruby193\bin;C:\Pro...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	AMD64

(…paragraph below not needed…)

~~Install the Ruby Development Kit:~~

~~http://rubyinstaller.org/downloads/~~

~~This can be installed in a directory of your choice; e.g., C:\RubyDevKit.~~

Verify your ruby installed version by typing the following at the command line:

```
ruby --version
```

The response should be similar to:

```
ruby 2.0.0p481 (2014-05-08) [i386-mingw32]
```

Ensure your Ruby installation knows where to find additional downloadable modules by typing the following at the command line:

```
gem sources –a http://rubygems.org/
```

If interested, explore these Ruby tutorials:
https://www.ruby-lang.org/en/documentation/quickstart/
http://www.tutorialspoint.com/ruby/ruby_overview.htm

WebDriver

Install the ruby webdriver library by typing the following at the command line:

```
gem install selenium-webdriver
```

(Note the “gem” command is provided by the ruby installation.)

If you are working with Firefox

No special steps are necessary for Firefox.

If you are working with Chrome

Install a ruby library for using webdriver with Chrome by typing the following at the command line:

```
gem install chromedriver-helper
```

Download the chromedriver executable for either Windows or Mac:

http://chromedriver.storage.googleapis.com/2.10/chromedriver_win32.zip

http://chromedriver.storage.googleapis.com/2.10/chromedriver_mac32.zip

After the zip file is downloaded, install the driver by extracting the file `chromedriver.exe` and placing it in your `ruby/bin` directory.

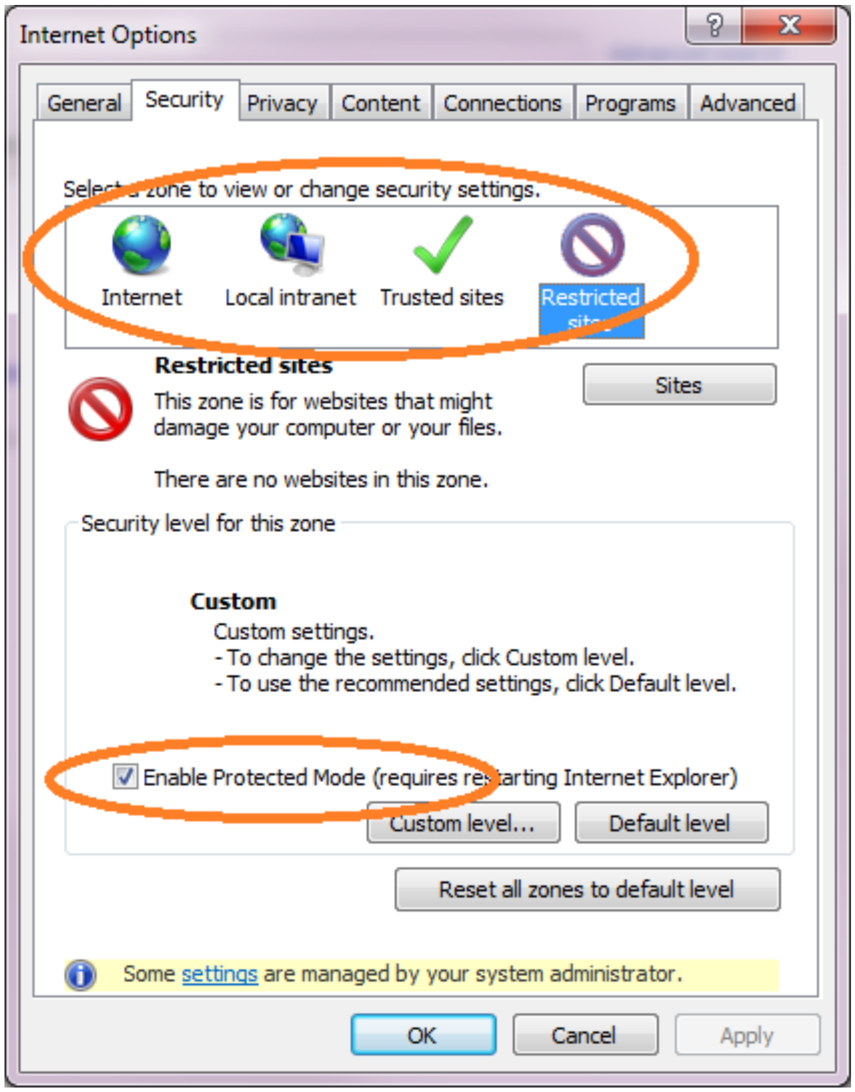
If you are working with IE

Download iedriverserver (either 32- or 64-bit version, as appropriate for your Windows version) from :

http://selenium-release.storage.googleapis.com/2.42/IEDriverServer_Win32_2.42.0.zip
or
http://selenium-release.storage.googleapis.com/2.42/IEDriverServer_x64_2.42.0.zip

After the zip file is downloaded, install the iedriverserver by extracting the file `IEDriverServer.exe` and placing it in your `ruby/bin` directory.

Configure IE security settings as follows. Inside IE, open Internet Options, then select the Security tab:



Select a zone to view or change security settings.

Restricted sites
This zone is for websites that might damage your computer or your files.

There are no websites in this zone.

Security level for this zone

Custom
Custom settings.
- To change the settings, click Custom level.
- To use the recommended settings, click Default level.

[✓] Enable Protected Mode (requires restarting Internet Explorer)

Some settings are managed by your system administrator.

Ensure that all zones have the same setting for the “Enable Protected Mode” checkbox. (Shouldn’t matter whether enabled or disabled, as long as all zones are the same).

If you are working with Safari

No special steps are necessary for Safari.

Additonal Information

Good documentation on WebDriver can be seen here:

http://docs.seleniumhq.org/docs/03_webdriver.jsp

Good information about using WebDriver from Ruby can be found here:
http://code.google.com/p/selenium/wiki/RubyBindings

RSpec

Install RSpec by typing the following at the command line:

```
gem install rspec
```

Verify your RSpec installed version by typing the following at the command line:

```
rspec --version
```

The response should be similar to:

```
3.0.0
```

Be sure you have RSpec version 3 or later.

Create a sample test – use any text editor to save the following text into a file called “shakespeare.rb”, in the directory of your choice:

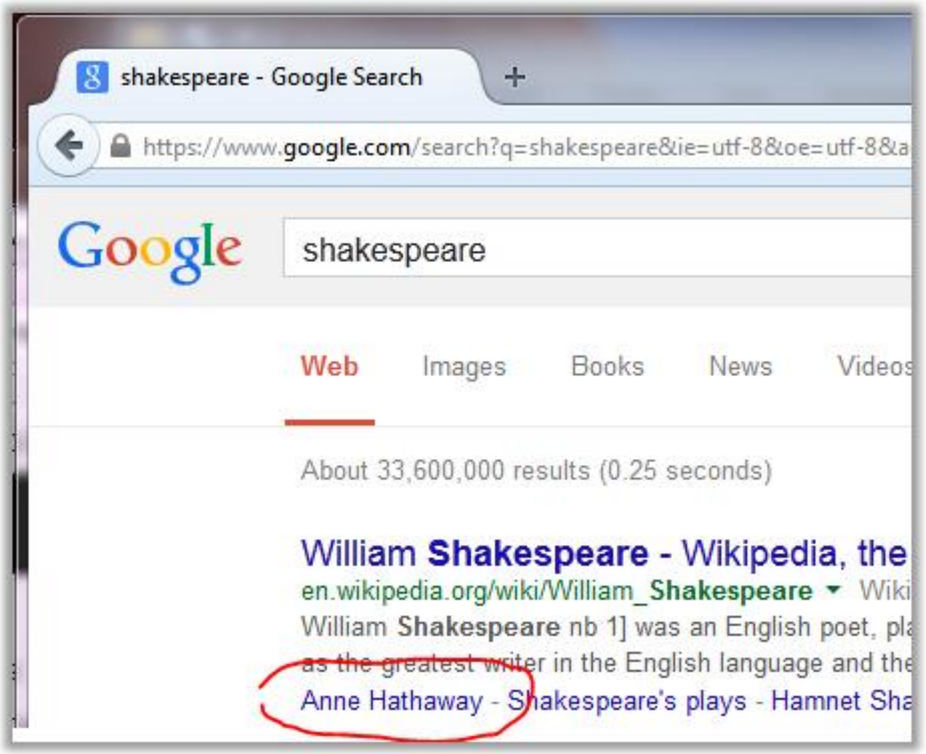
```
#  Open google.com, search for 'shakespeare', look for 'Anne Hathaway'

require 'rspec'
require 'selenium-webdriver'

browser = Selenium::WebDriver.for :ff

RSpec.configure do |config|
    config.before(:each) { @browser = browser }
    config.after(:suite) { browser.quit unless browser.nil? }
end

describe "Simple demo of google search: " do
    before(:each) do
        @browser.navigate.to("http://google.com/")
        sleep 1
    end

    context "When we have searched for shakespeare, " do
        before(:each) do
            @browser.find_element(:name, 'q').send_keys('shakespeare')
            @browser.find_element(:name, 'btnG').click
            sleep(3)
        end

        it "should contain Anne Hathaway" do
            expect( @browser.page_source ).to include("Anne Hathaway")
        end
    end
end
```

The above example will try to use Firefox. If you prefer Chrome, on line six change the "`ff`" to "`chrome`". If you prefer Internet Explorer, change the "`ff`" to "`ie`". If you prefer Safari, change the "ff" to "safari".

Type the following at the command line:

```
rspec shakespeare.rb
```

Your browser should open and end up showing a results page like so:



shakespeare - Google Search

https://www.google.com/search?q=shakespeare&ie=utf-8&oe=utf-8&a

shakespeare

About 33,600,000 results (0.25 seconds)

William Shakespeare - Wikipedia, the

en.wikipedia.org/wiki/William_Shakespeare ▾ Wiki

William Shakespeare nb 1] was an English poet, pla

as the greatest writer in the English language and the

Anne Hathaway - Shakespeare's plays - Hamnet Sha

And your console window should show the following:

```
C:\>rspec shakespeare.rb
.

Finished in 6.96 seconds (files took 8.02 seconds to load)
1 example, 0 failures
```

For more information about RSpec, refer to the following tutorials and book:

- http://code.tutsplus.com/tutorials/ruby-for-newbies-testing-with-rspec--net-21297
- http://blog.teamtreehouse.com/an-introduction-to-rspec
- http://pragprog.com/book/achbd/the-rspec-book

Capybara

Install the ruby capybara library by typing the following at the command line:

```
gem install capybara
```

Create a sample test – use any text editor to save the following text into a file called “shakespeare2.rb”:

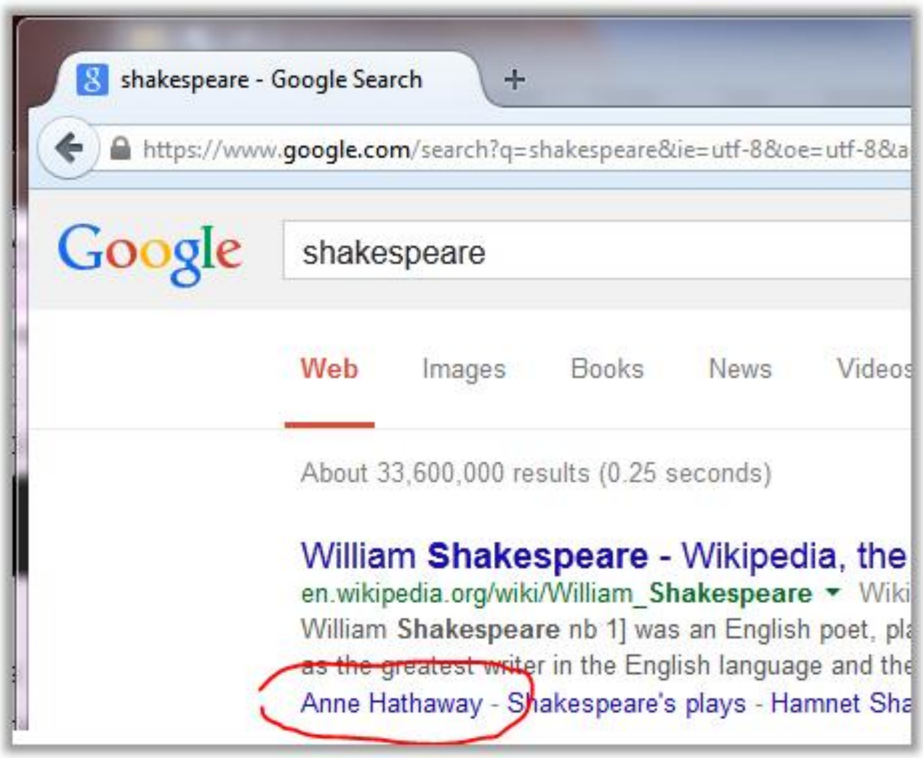
```
#  Open google.com, search for 'shakespeare', look for 'Anne Hathaway'

require 'capybara'
require 'selenium-webdriver'
require 'capybara/rspec'

Capybara.default_driver = :selenium

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

RSpec.configure do |config|
    config.include Capybara::DSL
end

describe "Simple demo of google search: " do
    before(:each) do
        visit("http://google.com/")
        sleep 1
    end

    context "When we have searched for shakespeare, " do
        before(:each) do
            fill_in('q', :with => 'shakespeare')
            find(:css, '[name="btnG"]').click
            sleep(3)
        end

        it "should contain Anne Hathaway" do
            expect(page).to have_content('Anne Hathaway')
        end
    end
end
```

The above example will try to use Firefox. If you prefer Chrome, on line ten change the “`ff`” to “`chrome`”. If you prefer Internet Explorer, change the “`ff`” to “`ie`”. If you prefer Safari, change the “ff” to “safari”.

Type the following at the command line:

```
rspec shakespeare2.rb
```

Your browser should open and end up showing a results page like so:



https://www.google.com/search?q=shakespeare&ie=utf-8&oe=utf-8&a

shakespeare

About 33,600,000 results (0.25 seconds)

William **Shakespeare** - Wikipedia, the
en.wikipedia.org/wiki/William_**Shakespeare** ▾ Wiki
William **Shakespeare** nb 1] was an English poet, pla
as the greatest writer in the English language and the
Anne Hathaway - Shakespeare's plays - Hamnet Sha

And your console window should show the following:

```
C:\>rspec shakespeare2.rb
.

Finished in 6.96 seconds (files took 8.02 seconds to load)
1 example, 0 failures
```

For more information about Capybara, refer to the following tutorials and book:

- http://jnicklas.github.io/capybara/
- https://github.com/jnicklas/capybara
- http://www.amazon.com/Application-Testing-Capybara-Matthew-Robbins/dp/1783281251

RSpec output format

RSpec can generate output in a few different formats.

Default “Progress” Format

By default, it uses the “progress” formatter, which generates output in the form of an ASCII progress bar on the standard output. It looks like this:

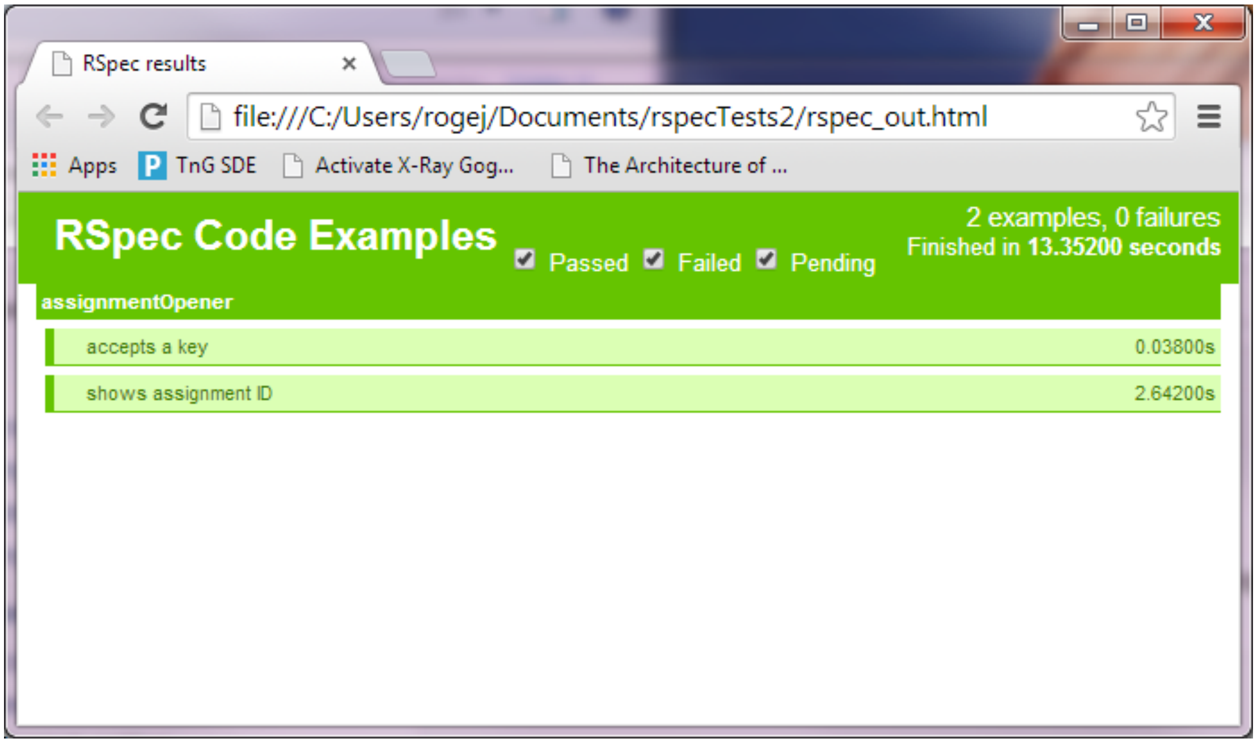
```
....F....F....
```

Eeach '`.`' represents a passing test, each 'F' is a failing test. So in the above example, there are a total of fourteen tests, with twelve successes and two failures. Were there only a single test, you would see only a single dot or a single 'F'.

<u>HTML Format</u>

RSpec can also generate output in HTML format. This is enabled by supplying the "`--format html`" command-line option.

Below is an example of HTML output format, where all (two of two) tests are passing:



RSpec Code Examples

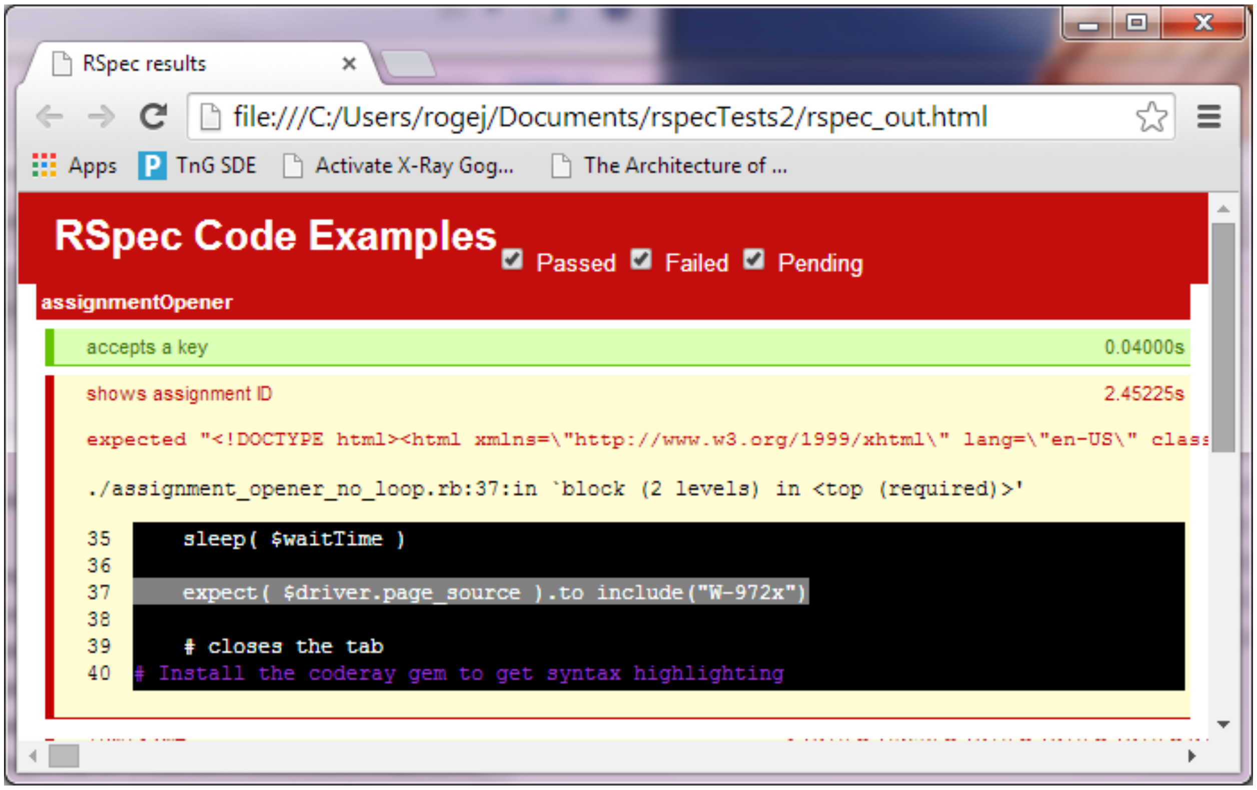
✓ Passed ✓ Failed ✓ Pending

2 examples, 0 failures
Finished in **13.35200 seconds**

assignmentOpener

```
accepts a key                                                    0.03800s
shows assignment ID                                              2.64200s
```

And here is an example of HTML output format from the same test, but where one of two tests is failing:



assignmentOpener

accepts a key 0.04000s

shows assignment ID 2.45225s

```
expected "<!DOCTYPE html><html xmlns=\"http://www.w3.org/1999/xhtml\" lang=\"en-US\" class
./assignment_opener_no_loop.rb:37:in `block (2 levels) in <top (required)>'
```

```
    sleep( $waitTime )

    expect( $driver.page_source ).to include("W-972x")

    # closes the tab
# Install the coderay gem to get syntax highlighting
```

Testing in Mobile Browsers using Appium

Appium provides an implementation of the WebDriver protocol for mobile browsers. Using Appium, you are able to take the same WebDriver automation scripts that work with desktop browsers, and run those scripts targeting Safari on a USB-tethered iPad or iPhone, or targeting Chrome on a USB-tethered Android device. On a Mac you can also target the iOS Simulator provided with the Xcode development tools.

The test scripts can be written using Ruby, WebDriver, RSpec, and Capybara – in addition to other tools.

Appium is available for either Mac or Windows. On Windows it can be used to test USB-tethered Android devices. On Mac it can be used to test USB-tethered iOS devices and Android devices.

The Appium application is available in two forms:

- An app written in Node.js, and run with the “node” command.
- A packaged GUI application that bundles everything required, including Node.js.

Either form can be used. The version run with the “node” command is about 10% faster, and the packaged GUI app can consume significant memory.

Follow the steps below to configure a Mac for testing on tethered iOS devices using Ruby, WebDriver, RSpec, Capybara, and Appium.

> *(Much of the following summarized from:*
> *https://github.com/appium/ruby_console/blob/master/osx.md*
> *Refer to that document for more detailed steps.)*

Ensure your Mac is running OS X 10.9.2 or better.

Install Xcode 5.1.1. (Do not use an older version.)

Install Xcode command-line tools. (Xcode → Open Developer Tools → More Developer Tools). See also:
(http://docwiki.embarcadero.com/RADStudio/XE4/en/Installing_the_Xcode_Command_Line_Tools_on_a_Mac)

Install Java 7 from here:
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Download and install Ruby 2.1.1 from here:
https://www.ruby-lang.org/en/downloads/
Note that you may be directed to another site: rubyinstaller.org.

For more details, refer to the earlier section on installing ruby.

Update RubyGems and Bundler

```
$ gem update --system ;\
$ gem install --no-rdoc --no-ri bundler ;\
$ gem update ;\
$ gem cleanup
```

Ensure RubyGems version is >= 2.1.5

```
$ gem --version
2.1.5
```

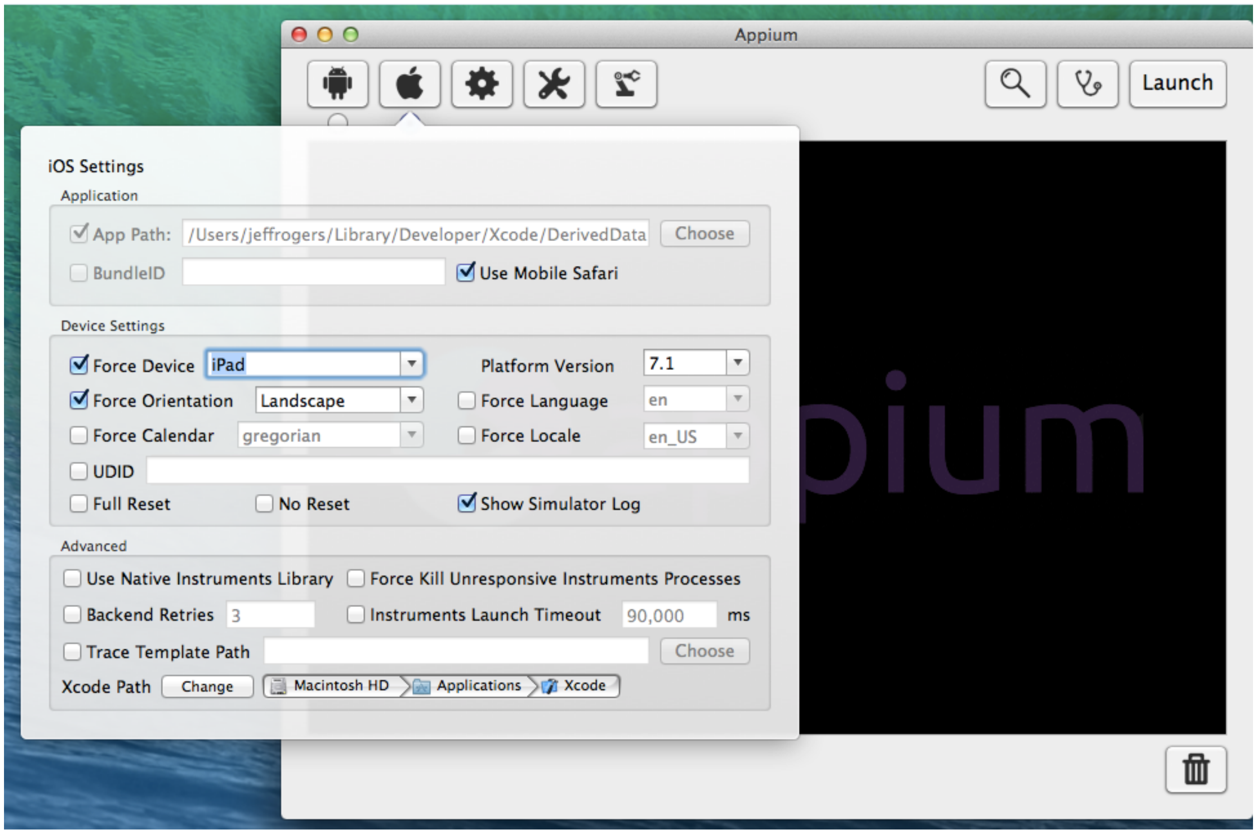
Install appium_lib & appium_console gems

```
$ gem uninstall -aIx appium_lib ;\
$ gem uninstall -aIx appium_console ;\
$ gem install --no-rdoc --no-ri appium_console
```

Download and Install the Appium Mac app from http://appium.io.

Launch Appium.

Configure Appium as shown below.



iOS Settings

Application

[✓ App Path:] /Users/jeffrogers/Library/Developer/Xcode/DerivedData Choose
BundleID [✓ Use Mobile Safari]

Device Settings

[✓ Force Device] iPad Platform Version 7.1
[✓ Force Orientation] Landscape Force Language en
Force Calendar gregorian Force Locale en_US
UDID
Full Reset No Reset [✓ Show Simulator Log]

Advanced

Use Native Instruments Library Force Kill Unresponsive Instruments Processes
Backend Retries 3 Instruments Launch Timeout 90,000 ms
Trace Template Path Choose
Xcode Path Change Macintosh HD > Applications > Xcode

Start the Appium server by clicking the “Launch” button.

Create a sample test – use any text editor to save the following text into a file called “shakespeare3.rb”:

```
#  Open google.com, search for 'shakespeare', look for 'Anne Hathaway' in result

require 'capybara'
require 'capybara/dsl'
require 'selenium-webdriver'
require 'capybara/rspec'
require 'appium_lib'

Capybara.register_driver :ipad do |app|
    caps = {
        :platform => 'Mac',
        :deviceName => 'iPad',
        :platformName => 'iOS',
        :platformVersion => '7.1',
        :app => 'safari'
    }
    Capybara::Selenium::Driver.new( app, {
        :browser => :remote,
        :url => "http://localhost:4723/wd/hub/",
        :desired_capabilities => caps
    })
end
Capybara.default_driver = :ipad
```

```
RSpec.configure do |config|
    config.include Capybara::DSL
end

describe "Simple demo of google search: " do
    before(:each) do
        visit "http://google.com/"
        sleep 1
    end

    context "When we have searched for shakespeare, " do
        before(:each) do
            fill_in('q', :with => 'shakespeare')
            begin
                click_button('Search')   # mobile
            rescue
                find(:id, 'tsbb').click  # mobile-simulator
            end
           sleep(3)
        end

        it "result contains Anne Hathaway" do
            expect(page).to have_content('Anne Hathaway')
        end
    end
end
```

Type the following at the command line:

```
rspec shakespeare3.rb
```

The iOS simulator should launch, and inside that mobile safari should launch. It should go to google.com, search for "shakespeare", and then show the result briefly. Then the simulator will close. Your console window should show something like the following:

```
$ rspec shakespeare3.rb
.

Finished in 34.96 seconds (files took 0.42163 seconds to load)
1 example, 0 failures
```

To test on a real iOS device

Install ios_webkit_debug_proxy on your Mac, from:
    https://github.com/google/ios-webkit-debug-proxy

Get new version of SafariLauncher.zip from:
    https://dl.dropboxusercontent.com/u/72075315/Appium.zip

Get & save the UDID of your iOS device:

    (Adapted from: http://bjango.com/help/iphoneudid/)

Every iPhone, iPod touch and iPad has a unique identifier number associated with it, known as a UDID (Unique Device ID). Your UDID is a 40-digit sequence of letters and numbers that looks like this: 0e83ff56a12a9cf0c7290cbb08ab6752181fb54b. It's common for developers to ask for your UDID as they require it to give you beta copies of iOS apps.

<u>Finding your UDID using iTunes</u>

- Open iTunes (the Mac or PC app, not iTunes on your iPhone).
- Plug in your iPhone, iPod touch or iPad.
- Click its name under the devices list.
- Ensure you're on the Summary tab.
- Click on the text that says Serial Number. It should change to say Identifier (UDID).
- Select Copy from the Edit menu.
- Your UDID is now in the clipboard, so you can paste it into an email or message.

Ensure your iOS device is connected to your Mac via USB cable.

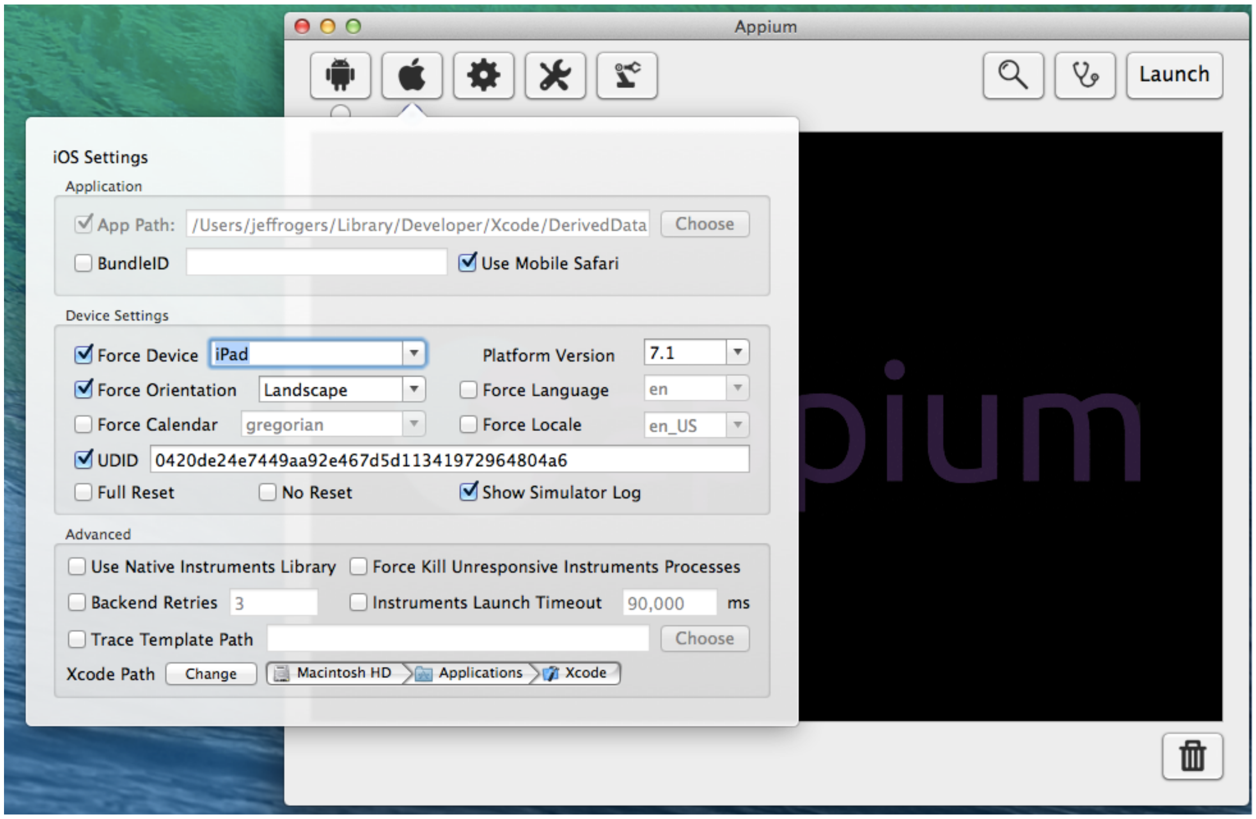
Ensure `ios_webkit_debug_proxy` is running on mac on port 27753, using the following command:

```
$ ios_webkit_debug_proxy -c <udid>:27753 -d
```

where `<udid>` is the UDID of your iOS device.

Stop the Appium server by clicking the "Stop" button.

Configure Appium as shown below. (Note: only the UDID checkbox and field differ from the previous screen shot.) Fill in the UDID from your iOS device.



iOS Settings

Application

☑ App Path: /Users/jeffrogers/Library/Developer/Xcode/DerivedData Choose

☐ BundleID ☑ Use Mobile Safari

Device Settings

☑ Force Device: iPad — Platform Version: 7.1

☑ Force Orientation: Landscape — ☐ Force Language: en

☐ Force Calendar: gregorian — ☐ Force Locale: en_US

☑ UDID: 0420de24e7449aa92e467d5d11341972964804a6

☐ Full Reset ☐ No Reset ☑ Show Simulator Log

Advanced

☐ Use Native Instruments Library ☐ Force Kill Unresponsive Instruments Processes

☐ Backend Retries: 3 ☐ Instruments Launch Timeout: 90,000 ms

☐ Trace Template Path Choose

Xcode Path: Change — Macintosh HD > Applications > Xcode

Start the Appium server by clicking the “Launch” button.

Run the same test as last time, by typing the following at the command line:

```
rspec shakespeare3.rb
```

On your tethered iOS device, you will see a program called “SafariLauncher” start, then it will switch to Safari. Safari will load and pause for a few seconds, then it should go to google.com, search for “shakespeare”, and then show the result briefly. Then the browser page will close. Your console window should show something like the following:

```
$ rspec shakespeare3.rb
.

Finished in 54.96 seconds (files took 0.42163 seconds to load)
1 example, 0 failures
```

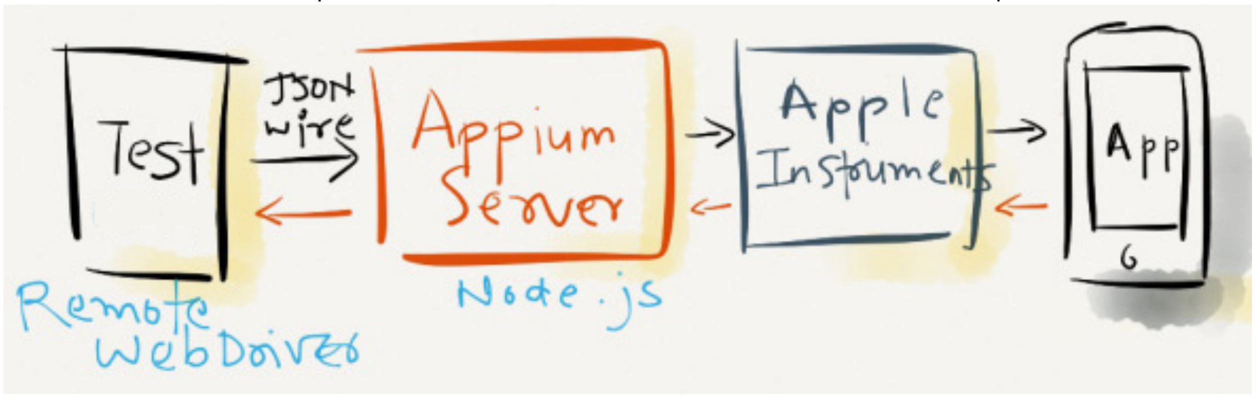
Android

To be added….

Using WebDriver Across a Network

When you have a webdriver instance (like Appium) running, you can control it from a separate machine on the network. For example, you can run Appium on a Mac, then run a ruby test on a PC, and have that ruby test talk to Appium on the Mac to control a browser on an iPad which is tethered to that Mac.

PC |←----------------------------- Mac ----------------------------→| iPad



Below are some ruby code fragments showing how to (a) initiate a test on a local browser, versus (b) initiate a test on a remote computer somewhere on the network. (The remote computer will in turn run the test on either [i] a device simulator, or [ii] a USB-tethered device, depending on how Appium is configured on that remote computer.)

Let's assume Appium is running on a Mac at IP address: **`10.88.90.122`**.

If you are using Capybara and RSpec:

```
# Local firefox
browser = Selenium::WebDriver.for :ff

# Remote safari
browser = Selenium::WebDriver.for :remote,
          :url => "http://10.88.90.122:4723/wd/hub",
          :desired_capabilities => :safari
```

If you are using RSpec without Capybara:

```
# Local firefox
Capybara::Selenium::Driver.new(app, :browser => :ff)

# Remote safari
Capybara::Selenium::Driver.new(app,
      :browser => :remote,
      :url => "http://10.88.90.122:4723/wd/hub",
      :desired_capabilities => :safari)
```

Executing tests containing either of these remote Ruby fragments will cause Appium on the Mac to launch a browser and then the Ruby code will drive that browser. Whether that browser launches on a tethered device, or in a simulator, depends on how Appium is configured on the Mac.

If two or more people try to run tests at the same time on the same remote machine, the first person wins. While a test is in progress, all other requests will fail with an error message that contains the following:

> ***A new session could not be created. (Original error: Requested a new session but one was in progress)***

Feature Testing in Pega's automated test environment

Integrating with Jenkins

To be added….

Integrating with Validation Engine

To be added….

Best Practices

Test Directory Structure

Recommended test directory structure shown below:

```
myTestDir/
      test_1.rb
      test_2.rb
      runtest.bat
      runtest.sh
      spec/
          spec_helper.rb
          shared_context_1.rb
          shared_context_2.rb
```

Descriptions:

`myTestDir` – holds test source files, test runner scripts, and subdirectories
`test_n.rb` – an RSpec test source file

`runtest.bat` & `runtest.sh` – scripts for Windows and Unix which can take in a variety of command-line arguments and assign them to environment variables, then execute tests via the “rspec” command.
`spec/` -- a directory to hold a variety of helper files.
`spec_helper.rb` – a helper file that is automatically included by RSpec in all tests executed.
`shared_context_n.rb` – an RSpec shared context file, to hold common utility methods

How Tests are Identified

Tests in RSpec can be identified in a number of different ways.

1. By source file and/or directory name
2. By line number within a source file
3. By test name
4. By tag

Generally the most useful mechanisms are test name and tag.

Consider an RSpec source file structured as shown below:

```
describe “feature under test” do
        context “in situation A” do
                it “meets acceptance criterion A1” do
                end
                it “meets acceptance criterion A2” do
                end
        end
        context “in situation B” do
                it “meets acceptance criterion B1” do
                end
                it “meets acceptance criterion B2” do
                end
        end
end
```

This source contains four distinct tests, each with a distinct name. Each test name is the concatenation of the strings that are supplied at each layer of the hierarchy: describe, context, and it. In this case the test names are:

```
feature under test in situation A meets acceptance criterion A1
feature under test in situation A meets acceptance criterion A2
feature under test in situation B meets acceptance criterion B1
feature under test in situation B meets acceptance criterion B2
```

Each test, or a group of tests, can be identified by its full name or any substring that occurs within the name. So “criterion A1” will identify exactly one test, and “situation B” will identify two tests.

Now, consider the same RSpec source file, but with some tags added, as shown below:

```
describe “feature under test” do
        context “in situation A” do
                it “meets acceptance criterion A1” do
                end
                it “meets acceptance criterion A2” :quarantine do
                end
        end
        context “in situation B” do
```

```
                it “meets acceptance criterion B1” :quarantine do
                end
                it “meets acceptance criterion B2” do
                end
        end
end
```

This source contains the same four tests, but now two of them are marked with the “`:quarantine`” tag. This provides a way for us to tell RSpec to exclude those tests from execution.

RSpec command-line options

The most useful command-line options are:

`--out myOutputFile.html`	to specify the name of the output file.
`--format html`	to generate HTML-formatted output.
`--example 'string'`	to run tests whose names contain “`string`”
`--tag ~myTag`	to exclude tests marked with “myTag”

In typical real-world usage, you will combine many options in a single command. For example, the command below will look for tests in a file called "myTestFile.rb", and will run those tests whose names contain the strings "shows" or "accepts", except those marked with the tag "quarantine". It will format the test output as HTML and will save that output in a file called "rspec_out.html".

```
rspec myTestFile.rb --example "shows" --example "accepts"
                    --tag "~quarantine" --format html --out rspec_out.html
```

Additional command-line options are described below.

To run all tests in all files in current working directory:

```
rspec .
```

To run tests in a specific file:

```
rspec myTest.rb
```

To run all tests in multiple files:

```
rspec file1.rb file2.rb
```

To run all tests in all files in multiple directories:

```
rspec path1 path2
```

To run all tests in all files in a specific directory:

```
rspec --default-path /my/desired/path/
```

To redirect output from STDOUT to a file:

```
rspec . --out myOutputFile.txt
```

To change format of output to list each test and result on a separate line:

```
rspec . --format documentation
rspec . --format html
```

To run tests in all files whose names match a pattern:

```
rspec --pattern "spec/**/*_spec.rb"
```

To run only some examples (i.e., describe/context/it strings), where the full nested name includes a certain string:

```
rspec . --example 'one string’
```

To run only some examples (i.e., describe/context/it strings), where the full nested name includes either of two specific strings:

```
rspec . --example 'one string' --example 'another string'
```

To run tests marked with a specific tag:

```
rspec . --tag myTag
```

To run tests marked with a specific tag which has a specific value:

```
rspec . --tag myTag:myValue
```

To exclude tests marked with a specific tag, use the tilde:

```
rspec . --tag ~myTag
rspec . --tag ~myTag:myValue
```

To run tests marked with myTag1 but exclude any also marked myTag2:

```
rspec . --tag myTag1 --tag ~myTag2
rspec . --tag myTag1:myValue1 --tag ~myTag2:myValue2
```

<u>More Best Practices</u>

Keep Descriptions Short

A spec description should never be longer than 40 characters. If this happens, it suggests you should split it using a context (some exceptions are allowed).

```
# bad
it "has 422 status code if an unexpected param will be added" do
  response.is_expected.to respond_with 422
end
```

```
# good
context "when not valid" do
  it { is_expected.to respond_with 422 }
end
```

Use Expect instead of Should syntax

```
# bad
it 'creates a resource' do
  response.should respond_with_content_type(:json)
end

# good
it 'creates a resource' do
  expect(response).to respond_with_content_type(:json)
end
```

Configure Rspec to only accept the new syntax on new projects, to avoid having the two syntaxes all over the place.

```
# good
# spec_helper.rb
RSpec.configure do |config|
  # ...
  config.expect_with :rspec do |c|
    c.syntax = :expect
  end
end
```

On one line expectations or with implicit subject we should use is_expected.to.

```
# bad
context 'when not valid' do
  it { should respond_with 422 }
end

# good
context 'when not valid' do
  it { is_expected.to respond_with 422 }
end
```

Use Contexts

Contexts are a powerful method to make your tests clear and well organized. In the long term this practice will keep tests easy to read.

```
# bad
it "should have 200 status code if logged in" do
  response.should respond_with 200
end
it "should have 401 status code if not logged in" do
  response.should respond_with 401
end

# good
```

```
context "when logged in" do
  it { should respond_with 200 }
end
context "when logged out" do
  it { should respond_with 401 }
end
```

(Over)use Describe and Context

Big specs can be a joy to play with as long as they are ordered and DRY. Use nested describe and context blocks as much as you can, each level adding its own specificity in the before block.

To check your specs are well organized, run them in 'nested' mode (spec spec/my_spec.rb -cf nested).

Using before(:each) in each context and describe blocks will help you set up the environment without repeating yourself. It also enables you to use it {} blocks.

```
# Bad:

describe User do

  it "should save when name is not empty" do
    User.new(:name => 'Alex').save.should == true
  end

  it "should not save when name is empty" do
    User.new.save.should == false
  end

  it "should not be valid when name is empty" do
    User.new.should_not be_valid
  end

  it "should be valid when name is not empty" do
    User.new(:name => 'Alex').should be_valid
  end

  it "should give the user a flower when gender is W" do
    User.new(:gender => 'W').present.should be_a Flower
  end

  it "should give the user a iMac when gender is M" do
    User.new(:gender => 'M').present.should be_an IMac
  end
end


# Good:

describe User do
  before { @user = User.new }

  subject { @user }

  context "when name empty" do
    it { should not be_valid }
    specify { @user.save.should == false }
  end

  context "when name not empty" do
```

```
    before { @user.name = 'Sam' }

    it { should be_valid }
    specify { @user.save.should == true }
  end

  describe :present do
    subject { @user.present }

    context "when user is a W" do
      before { @user.gender = 'W' }

      it { should be_a Flower }
    end

    context "when user is a M" do
      before { @user.gender = 'M' }

      it { should be_an IMac }
    end
  end
end
```

One Expectation Per Test

The "one expectation" tip is more broadly expressed as "each test should make only one assertion." This helps you on finding possible errors, going directly to the failing test, and to make your code readable.

Note: keep in mind that single expectation test does not mean single line test.

```
# bad
it "should create a resource" do
  response.should respond_with_content_type(:json)
  response.should assign_to(:resource)
end

# good
it { should respond_with_content_type(:json) }
it { should assign_to(:resource) }
```

Use subject

When you have several tests related to the same "subject" you can use the subject{} method to DRY them up. (DRY = Don't Repeat Yourself)

```
# bad
it { assigns("message").should match /The resource name is Genoveffa/ }
it { assigns("message").should match /it was born in Billyville/ }
it { assigns("message").creator.should match /Claudiano/ }

# good
subject { assigns("message") }
it { should match /The resource name is Genoveffa/ }
it { should match /it was born in Billville/ }
its(:creator) { should match /Claudiano/ }
```

Start contexts with "when" or "with"

Have you ever get a failed test with an incomprehensible error message like:

```
User non confirmed confirm email wrong token should not be valid
```

Start your contexts with "when" or "with" and get nice messages like:

```
User when non confirmed when confirm_email with wrong token should not be valid
```

Test Valid, Edge and Invalid cases

This is called Boundary value analysis, it’s simple and it will help you to cover the most important cases. Just split-up method’s input or object’s attributes into valid and invalid partitions and test both of them and there boundaries. A method specification might look like that:

```
describe "#month_in_english(month_id)" do
  context "when valid" do
    it "should return 'January' for 1" # lower boundary
    it "should return 'March' for 3"
    it "should return 'December' for 12" # upper boundary
  context "when invalid" do
    it "should return nil for 0"
    it "should return nil for 13"
  end
end
```

Run specs to confirm readability

Always run your specs with the ‘–format’ option set to ‘documentation’ (in RSpec 1.x the –format options are ‘nested’ and ‘specdoc’)

```
$ rspec spec/controllers/users_controller_spec.rb --format documentation


UsersController
  #create
    creates a new user
    sets a flash message
    redirects to the new user's profile
  #show
    finds the given user
    displays its profile
  #show.json
    returns the given user as JSON
  #destroy
    deletes the given user
    sets a flash message
    redirects to the home page
```

Continue to rename your examples until this output reads like clear conversation.

Formatting

Use ‘do..end’ style multiline blocks for all blocks, even for one-line examples. Further improve readability with a single blank line between all blocks and at the beginning and end of the top level #describe.

Again compare:

```
describe PostsController do
  describe '#new' do
    context 'when not logged in' do
      ...
      subject { response }
      it { should redirect_to(sign_in_path) }
      its(:body) { should match(/sign in/i) }
    end
  end
end
```

With the clearly structured code below:

```
describe PostsController do

  describe '#new' do
    context 'when not logged in' do
      ...

      it 'redirects to the sign in page' do
        response.should redirect_to(sign_in_path)
      end

      it 'displays a message to sign in' do
        response.body.should match(/sign in/i)
      end
    end
  end

end
```

A consistent formatting style is hard to achieve with a team of developers but the time saved from having to learn to visually parse each teammate’s style makes it worthwhile.

Mark Incomplete Tests as Pending

Often it is helpful to insert a placeholder into your test suite for tests which you know you will need, but have not yet written. RSpec calls these "pending" tests. This can be done in several ways.

It with no body

```
describe "an example" do
  it "is a pending example"
end
```

It containing a pending statement

```
describe "an example" do
  it "is implemented but waiting" do
    pending("something else getting finished")
    this_should_not_get_executed
  end
end
```

It replaced by xit

```
describe "an example" do
  xit "is pending using xit" do
    true.should be(true)
  end
end
```

Use the "Fail Fast" Option

You can often save a lot of time if you use the fail_fast option to tell RSpec to abort the run as soon as it encounters any failure.

```
RSpec.configure {|c|
    c.fail_fast = true
}
```

Identify tests (a.k.a. examples) to run by test name

Use the --example (or -e) option to filter the examples to be run by name.

The argument is compiled to a Ruby Regexp, and matched against the full description of the example, which is the concatenation of descriptions of the group (including any nested groups) and the example.

This allows you to run a single uniquely named example, all examples with similar names, all the example in a uniquely named group, etc, etc.

For example, suppose you have a file "mySpec.rb" containing:

```
describe "first group" do
  it "first example in first group" do; end
  it "second example in first group" do; end
end

describe "second group" do
  it "first example in second group" do; end
  it "second example in second group" do; end
end

describe "third group" do
  it "first example in third group" do; end
  context "nested group" do
    it "first example in nested group" do; end
    it "second example in nested group" do; end
  end
end

describe Array do
  describe "#length" do
    it "is the number of items" do
      Array.new([1,2,3]).length.should eq 3
    end
  end
end
```

Then the following commands should produce the output shown in the table below.

Command	Output should contain
`rspec . --example nothing_like_this`	`0 examples, 0 failures`
`rspec . --example example`	`7 examples, 0 failures`
`rspec . --example 'first example'`	`4 examples, 0 failures`
`rspec . --example 'first example in first group'`	`1 example, 0 failures`
`rspec . --example 'first group first example in first group'`	`1 example, 0 failures`
`rspec . --example 'first .* first example'`	`1 example, 0 failures`
`rspec . --example 'first group'`	`2 examples, 0 failures`
`rspec . --example 'second group first example'`	`1 example, 0 failures`
`rspec . --example 'third group'`	`3 examples, 0 failures`
`rspec . --example 'Array#length'`	`1 example, 0 failures`

More To be added….