Boosting developer’s productivity:

Better code quality with tests.

- Author: **Alexander Shvets**
- Role: Developer
- Year: **2013**

Goals of this presentation

- **Ruby language** can be used for other projects, not only for Ruby projects.
- **Acceptance tests** simulate human interactions with web applications. **Acceptance tests in ruby** can be used for supporting projects in other languages, such as Java.
- Tests are **very useful** to have, but they have maintenance cost. **Right strategy** should be selected for best results.
- Discuss **main concepts** and **ideas** in TDD/BDD world.
- Represent some **practical examples** of unit testing best practices.

Why Ruby?

- It is **scripting language** - no compilation required, convenient for writing automation scripts.
- It is very convenient for creating **working prototypes** of your future web application.
- It has powerful mechanism for creating new **Domain Specific Languages** (DSL):
  - **Rake** - DSL for build process.
  - **Sinatra** - DSL for describing simple web applications.
  - **Capybara** - DSL for acceptance tests.
  - **Capistrano** - deployment DSL.
  - **Chef** and **Puppet** - provisioning DSL.
- it has a lot of **gems** (libraries) for easy extending existing functionality.
- with the help of **JRuby** you can use complete power of existing Java libraries.
- it is **emerging standard**. Such companies as **github.com**, **twitter.com**, **groupon.com**, **hulu.com**, **livingsocial.com**, **yellowpages.com**, **vonage.com** already use it.
- Hosting services (PaaS), such as **Heroku**, **EngineYard**, **GoogleApps**, **Shelly Cloud** support it.

Why tests?

- to have an automated way to check if code is doing what it supposed to do (verify its correctness).
- to have an example of how to use code. You have to follow some formatting rules here.

How can we classify tests?

- by the **test scope**:
  - part of method or entire method;
  - entire class or group of classes;
  - entire system.
- by the **nature of interaction** between test and testing program:
  - direct code call;
  - simulation of user behavior: clicks, scrolls, opening/closing windows etc.).

Example: Cucumber script

- Approach: **direct code call** or **simulation of user behavior**

```
Scenario: Adding a subpage
  Given I am logged in
  Given a microsite with a home page

  When I press "Add subpage"
  And I fill in "Title" with "Gallery"
  And I press "Ok"

  Then I should see a document called "Gallery"
```

Exampe: RSpec script

- Approach: **direct code call** or **simulation of user behavior**

```
describe MyRubyClass do

  before :each do
    @my_ruby_class = MyRubyClass.new
  end

  it "should execute" do
    expect(@my_ruby_class.execute).to eql("success")
  end

end
```

Example: Capybara script

- Approach: **simulation of user behavior**

```
visit '/'

fill_in 'Login', :with => 'user@example.com'

fill_in 'Password', :with => 'mypassword'

click_button "Submit"
```

Types of tests

- **Unit test** - tests small piece of functionality in isolation (within single class).
- **Functional test** - tests various actions of a **controller** class.
- **Note 1**: **Controller** encapsulates interactions between classes (not just rails/struts/spring-web-flow controller).
- **Note 2**: Sometimes **functional** means that you create tests based on **functional requirements**.
- **Integration test** - test that hits **several layers of the system**, but not everything. Such tests describe the interaction among any number of controllers.
- **Acceptance test** - **end-to-end full-stack test** (everything). You simulate a browser, go through your controllers, hit the database and web services etc.
- **Note 3**: Sometimes acceptance tests are called as “functional”, “customer”, even “system” tests. And worse, for professional software testers it means something different.

Unit Tests

- try to answer: Do our objects do the right thing, are they convenient to work with?
- should be about **one particular feature**.
- everything it touches should be done **in memory**.
- **should not cross** some boundaries: access network, access database, use file system, call external services etc.
- you have to **stub or mock** dependencies that are hard to understand, initialize or manipulate.
- have to be **as simple as possible**, easy to debug, fast to execute.
- have to prove that created code **functions as intended** before it is used with other code.

Integration and Functional Tests

- **Unit Test Shortfall**: it **does not necessary work as expected** when you combine 2 units together, even if units work perfectly in isolation.
- try to answer: Does our code work against code we can't change?
- built by **combining the units of code** and testing that resulting combination works correctly.
- can do what **unit tests cannot**: access network or file system, read database, call external services etc..
- could be executed in **environment, close to production**.
- **Main disadvantage**: they touch more code, failures are harder to diagnose and maintain.

Acceptance Tests

- **Integration/Functional Test Shortfall**: it don't prove that a **complete feature works**.
- try to answer: Does the whole system work?
- created to **mirror the user stories**.
- utilize **user behavior simulation** approach.
- easily understood by **stakeholders**, **business analysts**, **testers**, and **developers**.

Relationships between different test types

- Writing **acceptance tests** tells us something about **how well we understand the domain**, but they **don't tell us how well we've written the code**.
- Running **acceptance tests** tells us about the **External Quality** of our system.
- Writing **unit tests** gives us a lot of feedback about the quality of our code, **Internal Quality**.
- Running **unit tests** tells us that we haven't broken any classes, but they don't give us enough confidence that the **system as a whole works**.
- **Integration tests** are filling the gap between acceptance and unit tests.

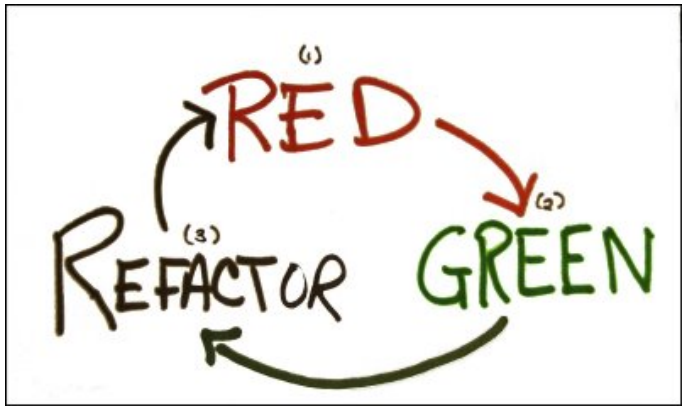
Test Coverage Strategies

- **Use all types of tests** - too expensive.
- **Unit tests only** - not enough.
- **Acceptance tests only** - good for small projects only.
- **Unit and acceptance tests** - good tradeoff, could be working solution.
- **Unit, acceptance and integration tests** - fits for most medium to large projects.

TDD vs. BDD

- Sometimes people think they (TDD, BDD) are different from each other. They are not.
- **TDD** is about writing test **before** you write code (tool).
- **BDD** is about providing **new behavior** (goal). TDD (tool) can be used to achieve this goal.

Red/Green/Refactor Cycle: TDD Concept



Steps of Red/Green/Refactor cycle

- Red - Write new, very small test for the code. Run it - it should fail - because code does not esxist yet.
- Green - Try to write just enough code to make that test to pass.
- Refactor - Now you make changes inside your code. They should not break existing test(s).
- Repeat - Add new tests, repeating whole process (Red-Green-Refactor) again and again.

Refactoring as Process

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. **Refactoring: Improving the Design of Existing Code**:

- Refactoring could be described as:
  - a **change** made to the **internal structure** of software to make it
    - easier to understand and cheaper to modify without changing its **observable behavior**.

Given-When-Then triad: BDD Concept

Description of Given-When-Then triad

- With TDD we are trying to understand **what an object is** instead of what **an object does**.
- If you change **internals only** (implementation) without changing **interface** (behavior):
  - test that depends on internals will fail;
  - test that depends on behavior - will not fail.
- BDD introduces the language we use to describe testing scenarios based on **Given**, **When**, **Then** words.
- Given some context, When some event occurs, Then I expect some outcome.
- Given I have potato, When I cook it, Then I can eat it.

Consequences of using TDD/BDD

- You think about **problem first**, before you write any line of code.
- Testing is **no longer** just about keeping defects from the users; instead, it's about helping the team **to understand the features** that the users need and to deliver those features reliably and predictably.
- Tests require **fair amount of efforts** to maintain/upgrade them.
- After applying TDD/BDD technics iteratively, your system will have more **modular architecture**, well-defined components with **published interfaces**.
- Instead of running **full-fledged application server with web application** you can run isolated test only. Thus, you can concentrate on one aspect of your code - reducing time spend for implementation or bug fix.

Common recommendations for writing tests

- Test with **a lot of dependencies and setup code** indicates imperfect architecture decisions in past. You need to start rearranging your code by minimizing/optimizing dependencies.
- Methods that are **too long** and as result - too hard to cover - require rearrangement/refactoring.
- Methods that are **slow** could be a symptom, not a cause.
- Classes that have **too many tests**, needs to be broken into smaller classes.
- If you see **two tests that contradict each other**, you have inconsistency in your specification/story.

RSpec

What is it?

- **Domain Specific Language** for describing tests written in Ruby.
- It uses such language words:
  - **describe**
  - **it** (example)
  - **expect**
  - **mock**
  - **should**
- Red/Green/Refactor concept used a lot with it.

Example: rspec script

```
class Bowling
  def hit(pins) end

  def score ; 0 end
end

require 'rspec'

describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

- Run it:

```
rspec bowling_spec.rb
```

- Result:

```
Bowling#score .

Finished in 0.00436 seconds
1 example, 0 failures
```

Cucumber

What is it?

- **Domain Specific Language** for describing tests written in Ruby.
- it uses **Gherkin** language (simplified plain English) for test description.
- Tests are written in terms of **features**.
- Each feature consists of one or more **scenarios**.
- Each scenario consists of one or more **steps**.
- Each step uses **Given**, **When**, **Then**, **And** and **But** words for describing what needs to be done.
- Can be used for **all types of tests**, not only for acceptance tests.
- Can be used for **other languages** like Java, C# or Python. You use ruby directly or search for Cucumber adapted for a given language.

Installation

- Install required gem:

```
gem 'cucumber-rails', :require => false
```

- or, if you don’t need rails part:

```
gem 'cucumber'
```

- All cucuber tests are located in **features** folder. Usually you have to create **features/support/env.rb** file where you keep configuration of cucumber:

```
# env.rb

require 'cucumber'
```

Example: Google Search

- You create your script in Gherkin:

```
Feature: Using Google

  Background: within google.com context
    Given I am within google.com

  Scenario: Searching for a term without submit

    Given I am on google.com
    When I enter "Capybara"
    Then I should see css "div#res li"

  @selenium
  Scenario: Searching with selenium for a term with submit

    Given I am on google.com
    When I enter "Capybara"
    And click submit button
    Then I should see results: "Capybara"

  @webkit
  Scenario: Searching with webkit for a term with submit

    Given I am on google.com
    When I enter "Capybara"
    And click submit button
    Then I should see results: "Capybara"
```

Example: Google Search (continued)

- For each Gherkin phrase create corresponding **step definition**:

```
Given(/^I am within google\.com$/) do
  Capybara.app_host = "http://google.com"
end

Given /^I am on google\.com$/ do
  visit('/')
end

When /^I enter "([^"]*)"$/ do |term|
  fill_in('q',:with => term)
end

Then /^I should see css "([^"]*)"$/ do |css|
  page.should have_css(css)
end

When(/^click submit button$/) do
   if Capybara.current_driver == :selenium
     find("#gbqfbw button").click
   else
     has_selector? ".gsfs .gssb_g span.ds input.lsb", :visible => true # wait for ajax to be finished

     button = first(".gsfs .gssb_g span.ds input.lsb")

     button.click
   end
end

Then(/^I should see results: "([^"]*)"$/) do |string|
  page.should have_content(string)
end
```

Example: Google Search in other language

- You can write cucumber tests in other languages, e.g.in Russian:

```
# language: ru

Функционал: Используем Гугл

  Сценарий: Поиск без посылки формы
    Дано I am on google.com
    Когда I enter "Capybara"
    Тогда I should see css "div#res li"

  @selenium
  Сценарий: Searching with selenium for a term with submit

    Дано I am on google.com
    Когда I enter "Capybara"
    И click submit button
    Тогда I should see results: "Capybara"

  @webkit
  Сценарий: Searching with webkit for a term with submit

    Дано I am on google.com
    Когда I enter "Capybara"
    И click submit button
    Тогда I should see results: "Capybara"
```

Example: Proteus Subscribe

- **Cucumber script** for proteus project:

```
Feature: Subscribe to Vonage phone service through Triton UI
  In order to subscribe to Vonage phone service
  As a user
  I want to fill in an order and submit

  @webkit
  Scenario: Typical Direct Flow
    When I go to Triton default landing page
    Then I should be on Plan Setup page

    When I select Plan
    And I select Phone Adapter
    Then I should see phone adapter message
    And I pick a new phone number
    And I click "Continue"
    Then I should be on Contact Information Page
```

Example: Proteus Subscribe (continued)

- **Cucumber steps** for proteus project:

```
Given(/^I am within localhost$/) do
  Capybara.app_host = "http://localhost:3000"
end

When /^I go to (.+)$/ do |page_name|
  visit path_to(page_name)
end

Then(/^I should be on Plan Setup page$/) do
  page.should have_content 'Select one of our most popular plans'
end

When(/^I select Plan$/) do
  find('.first.plan_column .plan:first-child a.save').click
end

When(/^I select Phone Adapter$/) do
  device_selector = '.equipment.section .adapter:first-child input[type=radio]'
  page.has_selector? device_selector, :visible => true # wait for ajax to be finished
  find(device_selector).click
end

Then(/^I should see phone adapter message$/) do
  page.should have_content 'Transfer your phone number or get a new one'
end
```

Example: Proteus Subscribe (continued)

```
When(/^I pick a new phone number$/) do
  find('input[value=new_did]').click
  select 'New Jersey', from: 'did_request[state_code]'
  select '732', from: 'did_request[npa]'
  select 'Red Bank 732-678-xxxx', from: 'did_request[nxx]'
  select '732-678-1231', from: 'did_request[selected_number]'
end

When(/^I click "Continue"$/) do
  find('.back_fwd_btns .next a').click
end

Then(/^I should be on Contact Information Page$/) do
  page.should have_content "Enter your contact information"
end
```

How to run

- When you have test, you can run it:

```
cucumber features/google_search.feature
cucumber features/direct_flow.feature
```

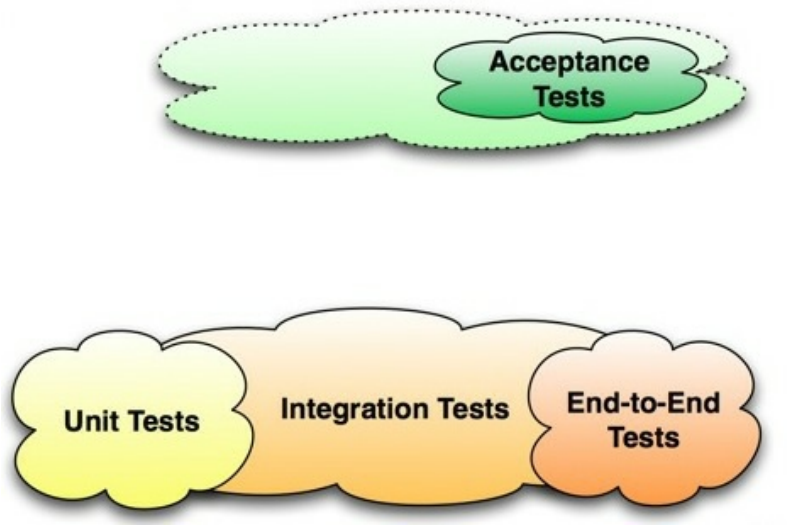
- Output:

```
Environment: test
Feature: Using Google

  Scenario: Searching for a term without submit # features/google_search.feature:3
    Given I am on google.com                   # features/step_definitions/google_steps.rb:5
    When I enter "Capybara"                    # features/step_definitions/google_steps.rb:9
    Then I should see css "div#res li"         # features/step_definitions/google_steps.rb:13
  @selenium
  Scenario: Searching with selenium for a term with submit # features/google_search.feature:10
    Given I am on google.com                               # features/step_definitions/google_steps.rb:5
    When I enter "Capybara"                                # features/step_definitions/google_steps.rb:9
    And click submit button                                # features/step_definitions/google_steps.rb:17
    Then I should see results: "Capybara"                  # features/step_definitions/google_steps.rb:29
  @webkit
  Scenario: Searching with webkit for a term with submit # features/google_search.feature:18
    Given I am on google.com                             # features/step_definitions/google_steps.rb:5
    When I enter "Capybara"                              # features/step_definitions/google_steps.rb:9
    And click submit button                              # features/step_definitions/google_steps.rb:17
    Then I should see results: "Capybara"                # features/step_definitions/google_steps.rb:29

11 steps (11 passed)
0m9.280s
```

Automated Acceptance Tests



Automated Acceptance Tests: Why?

- It’s **not possible or extremely expensive** to manually test each feature of your application.
- Making testing automated will **save a lot of time/money** for the company.
- In case of **adding new functionality** you want to know whether it breaks existing flow or not. Unit or integration tests do not know about new functionality and cannot fail on it yet.

Existing solutions for Acceptance Tests

- WebRat
- Watir
- Mechanize
- Selenium
- Capybara

Existing solutions based on Selenium

- Appium - test automation tool for native and hybrid mobile apps on iOS, Android, and FirefoxOS platforms
- Selendroid - test automate native or hybrid Android apps with Selendroid.
- Calabash IOS - lets you run Cucumber features on your IOS device or emulator.
- Calabash Android - lets you run Cucumber features on your Android device or emulator.
- iPhone Driver - allows testing on a UIWebView (a webkit browser accessible for 3rd party applications) on the iphone. It works through the use of an iphone application running on your iphone, ipod touch or iphone simulator.
- ios-driver - IOS automation for native, hybrid and mobile web.

Capybara

What is it?

- Domain Specific Language for describing **acceptance tests** written in Ruby.
- **Simulates real user interactions** with web application.
- Provides **higher level API** to user interaction. You can have or write drivers that fit into this API. Example: **selenium** or **webkit** driver.
- Provides **auto-waiting** feature - powerful synchronization for asynchronous processes, like AJAX.
- It could be used inside both **rspec** and **cucumber**.

Choices for writing acceptance tests

- selenium-client (deprecated).
- selenium-webdriver in Java.
- selenium-webdriver in Ruby.
- Capybara test with selenium driver.
- Capybara test with webkit driver.

Example 1: Java with selenium-webdriver

```
import org.openqa.selenium.*;
import org.openqa.selenium.support.ui.*;
import org.openqa.selenium.firefox.*;

WebDriver driver = new FirefoxDriver();

driver.get("http://www.google.com");

WebElement element = driver.findElement(By.name("q"));
element.sendKeys("Cheese!");

Wait<WebDriver> wait = new WebDriverWait(driver, 30);

wait.until(new ExpectedCondition<Boolean>() {
  public Boolean apply(WebDriver webDriver) {
    System.out.println("Searching ...");
    return webDriver.findElement(By.id("resultStats")) != null;
  }
});

element.submit();
```

- Too verbose.
- Is not script (require compilation, packaging, writing run script).
- Does not support smart auto-waiting feature.

Example 2: Ruby with selenium-webdriver

```
require 'selenium-webdriver'

driver = Selenium::WebDriver.for(:firefox)

driver.navigate.to('http://www.wikipedia.org')
expect(driver.find_element(:id, 'www-wikipedia-org')).
  not_to be_nil

driver.find_element(:id, 'searchInput').send_keys("iphone")
driver.find_element(:name, 'go').click

wait = Selenium::WebDriver::Wait.new(:timeout => 60) # seconds

wait.until {
  element = driver.find_element(:id, 'content')

  element.attribute("disabled").nil? ? true :
    element.attribute("disabled")
}

expect(driver.element.text).to match /iPhone/

driver.quit
```

- Require running separate selenium server (could be sometimes unacceptable)
- Produced code is not generic enough

Example 3: Capybara script for Proteus

```
require "capybara/dsl"

Capybara.app_host = 'http://localhost:3000'
Capybara.default_wait_time = 15

Capybara.current_driver = :selenium

include Capybara::DSL

visit "/triton?partner=VONAGE"

page.should have_content 'Select one of our most popular plans'

# select_plan
find('.first.plan_column .plan:first-child a.save').click

# select_phone_adapter
device_selector =
  '.equipment.section .adapter:first-child input[type=radio]'

# wait for ajax to be finished
page.has_selector? device_selector, :visible => true

find(device_selector).click

page.should have_content
  'Transfer your phone number or get a new one'
```

Example 3: Capybara script for Proteus (continued)

```
# select_new_number

find('input[value=new_did]').click

select 'New Jersey', from: 'did_request[state_code]'
select '732', from: 'did_request[npa]'
select 'Red Bank 732-678-xxxx', from: 'did_request[nxx]'
select '732-678-1231', from: 'did_request[selected_number]'


# click_next
find('.back_fwd_btns .next a').click

page.should have_content "Enter your contact information"
```

- Require running separate selenium server (could be sometimes unacceptable)
- Produced code is generic enough - it uses Capybara generic API

Example 4: Capybara script for MOLA

```
Capybara.app_host = "http://localhost:8080"
Capybara.default_wait_time = 15

Capybara.current_driver = :selenium

include Capybara::DSL

start_page = "/m/index-vdev.html"

visit start_page

page.should have_content "CUSTOMER ACCOUNT SIGN IN"

fill_in 'username', with: 'mola2'
fill_in 'password', with: 'test123'

find(:css, '.login').click

page.should have_content "Please tap here to view all action"

find(:xpath, ".//a[@href='#/home/extensions']").click

page.should have_content "Features"
```

Headless mode

- **capybara-webkit** gem lets you run same script in **headless mode**.
- It requires/uses **Qt** for rendering.
- It can execute **javascript** code.
- It is **faster** than selenium test.
- You don’t have to run **separate selenium server**.
- You don’t have to open **browser window** in order to execute tests.

Running in headless mode

- Install **capybara-webkit** gem:

```
gem install capybara-webkit
```

- Now you can run same script in headless mode by changing only driver name:

```
Capybara.current_driver = :webkit
```

Jasmine

What is it?

- It is simple framework for writing unit tests for **javascript code**.
- It works **similar to selenium** tests.
- It **has server** that builds and executes jasmine scripts.
- You access and run jasmine tests **from the browser**.
- **Reload** web page in the browser every time you do **changes in tests**.

Installation

- Install jasmine gem and init it:

```
$ gem install jasmine
```

```
$ jasmine init
```

- It will create **public/javascripts** and **spec/javascripts** folders with sample javascript class and corresponding unit test.

Configuration

- Correct **spec/javascript/support/jasmine.yml** file to point to correct location of your javascripts:

```
src_files, e.g.:
    - public/javascripts/MyJSCode.js
```

Example: Jasmine Test vs. RSpec

Jasmine

```
function MyJsClass() {}

MyJsClass.prototype.execute =
  function() { return "success"; };


describe("MyJsClass", function() {
  var myJsClass;

  beforeEach(function() {
    myJsClass = new MyJsClass();
  });

  it("should execute", function() {
    expect(myJsClass.execute()).
      toEqual("success");
  });
});
```

RSpec

```
require 'rspec'

class MyRubyClass
  def execute; "success" end
end

describe MyRubyClass do
```

```
  before :each do
    @my_ruby_class = MyRubyClass.new
  end

  it "should execute" do
    expect(@my_ruby_class.execute).
        to eql("success")
  end
end
```

Usage

- Run tests server:

```
$ rake jasmine
```

- and then access tests server from the browser:

```
open http://localhost:8888
```

Command line interface (selenium mode)

- You can run tests from command line:

```
$ rake jasmine:ci
```

- It will try to:
  - start tests server
  - launch tests in the browser (as selenium test)
  - display results in console:

```
>> Thin web server (v1.5.1 codename Straight Razor)
>> Maximum connections set to 1024Waiting for
jasmine server on 52937...

>> Listening on 0.0.0.0:52937, CTRL+C to stop
jasmine server started.
Waiting for suite to finish in browser ...
..........
```

Command line interface (webkit mode)

- **jasmine-headless-webkit** gem lets you run jasmine tests in **headless mode** with the help of **webkit driver**:

```
gem install jasmine-headless-webkit
```

- Now you can run all the tests:

```
jasmine-headless-webkit
rake jasmine:headless
```

- or selected tests:

```
jasmine-headless-webkit spec/javascripts/player_spec.js
```

- Result:

```
Running Jasmine specs...
...........
PASS: 11 tests, 0 failures, 0.018 secs.
```

Links

- [Growing Object-Oriented Software, Guided by Tests by Steve Freeman and Nat Pryce - www.growing-object-oriented-software.com/] (www.growing-object-oriented-software.com/)
- [The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends - http://www.amazon.com/RSpec-Book-Behaviour-Development-Cucumber/dp/1934356379] (http://www.amazon.com/RSpec-Book-Behaviour-Development-Cucumber/dp/1934356379)
- Cucumber Home - https://github.com/cucumber
- Rspec Home - http://rspec.info
- Capybara Home - https://github.com/jnicklas/capybara
- Jasmine - *http://pivotal.github.io/jasmine*
- [jasmine-headless-webkit gem - *http://johnbintz.github.io/jasmine-headless-webkit*] (http://johnbintz.github.io/jasmine-headless-webkit)
- [Why wait_until was removed from Capybara - *http://www.elabs.se/blog/53-why-wait_until-was-removed-from-capybara*] (http://www.elabs.se/blog/53-why-wait_until-was-removed-from-capybara)
- Ruby Bindings for Selenium - *http://code.google.com/p/selenium/wiki/RubyBindings*
- [rspec-rails and capybara 2.0: what you need to know - http://alindeman.github.io/2012/11/11/rspec-rails-and-capybara-2.0-what-you-need-to-know.html] (http://alindeman.github.io/2012/11/11/rspec-rails-and-capybara-2.0-what-you-need-to-know.html)
- Top 5 Sites Built with Ruby on Rails - http://www.bacancytechnology.com/blog/top-5-sites-built-with-ruby-on-rails
- Top 10 Sites Built with Ruby on Rails - http://blog.netguru.co/post/58995145341/top-10-

sites-built-with-ruby-on-rails

- 40 Sites Built with Ruby on Rails - http://designwebkit.com/inspiration/40-websites-built-with-ruby-on-rails/
- Improve your test readability using the xUnit structure - http://blog.plataformatec.com.br/2014/04/improve-your-test-readability-using-the-xunit-structure

Thank You!

Hola! # Спасибо! # 谢谢! # Toda!

Questions?

Suggestions?