Integrallis

RSpec & Capybara

BDD & Acceptance Testing in Rails

https://dl.dropboxusercontent.com/u/2968596/rspec_and_capybara.pdf

by Brian Sam-Bodden
@bsbodden

http://www.integrallis.com

Objectives

- Practical Behaviour-Driven Development
- Test-Driven Development with RSpec
- To become comfortable with the **common tools** used by Rubyists
- To learn and embrace the **practices** of successful Ruby/Rails developers

Requirements

What you'll need on your system to play along...

Requirements

- Git (http://git-scm.com/)
- RVM (https://rvm.io) [Optional]
- Ruby 1.9.3 (or 2.0)
- RubyGems
- Any code/text editor

Material Conventions
Part 1 - Ruby

- This 80 minute training consists of a mix of lecture time, guided exercises and some labs (in class if we have the time, take home if we don't)

FOLLOW ALONG

  - For the guided exercises you will see a green "follow along" sign on the slides

LAB 1.0

  - For the labs you'll see an orange sign with the lab number

Testing

Unit, Integration, Acceptance, BDD & TDD

Testing
A Means To An End

"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software"

http://agilemanifesto.org/principles.html

Testing
A Means To An End

- Testing traditionally done at the end of development “if time permitted” (hello waterfall)
- No language support or frameworks (back in the old days)
- Started from the “inside” with Unit Testing Frameworks
- Lots of well tested units, we were still left with a mess at the outer layers
- BDD came in to try to reverse the testing approach

http://c2.com/cgi/wiki?TenYearsOfTestDrivenDevelopment

Testing
A Means To An End

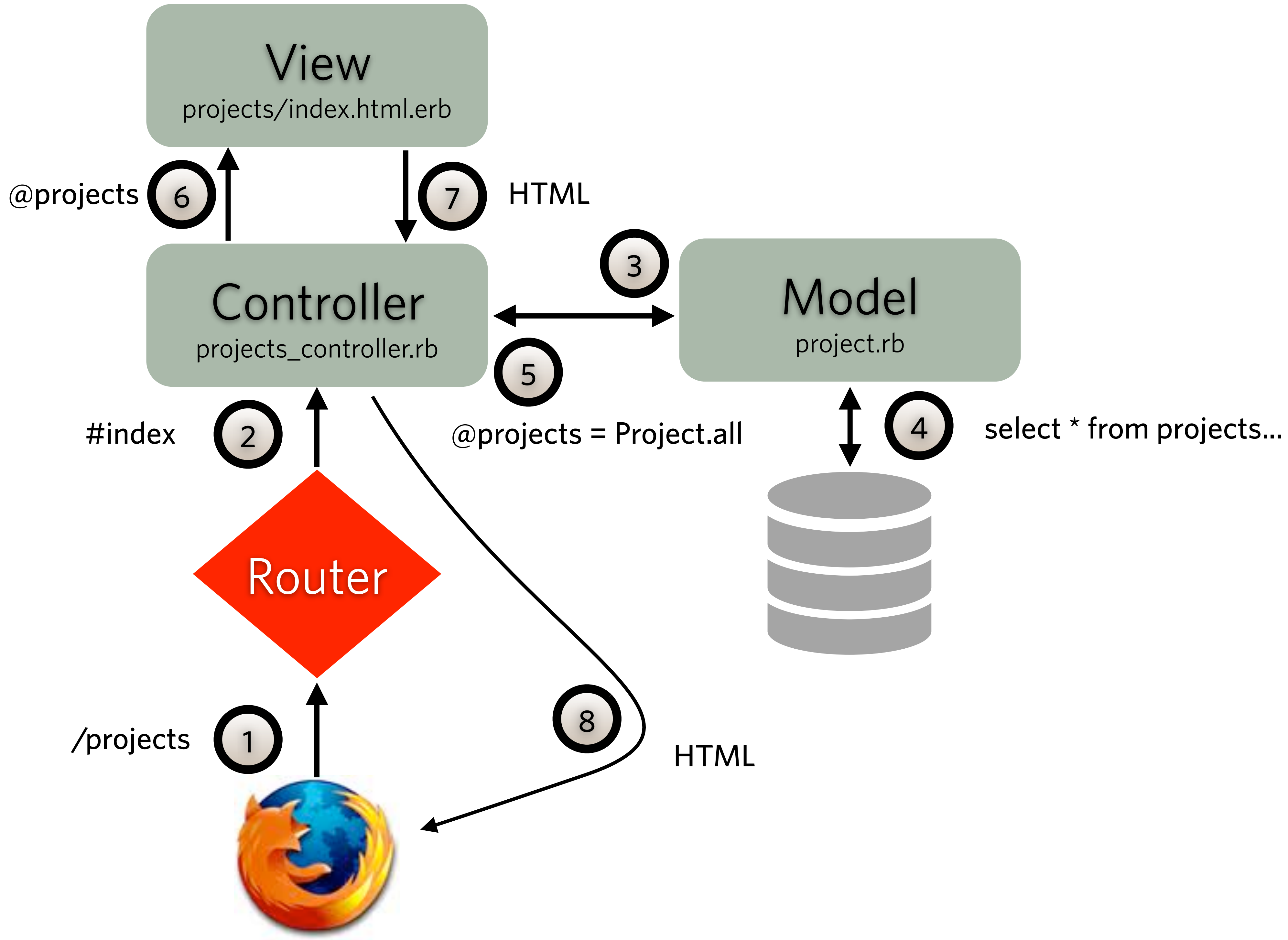
- BDD testing frameworks are DSLs (built on top of Unit Testing Frameworks) to “get the words rights”
- Most examples still use Units (class & methods) to teach BDD. Therefore developers still start at the inside.
- Rails showed early own that Web Application Testing CAN be automated
- Integration testing still hard to define for most developers
- Acceptance testing is NOT integration testing (unless you mean integrating with your users)

http://c2.com/cgi/wiki?TenYearsOfTestDrivenDevelopment

Request Handling

The Request-Response Pipeline

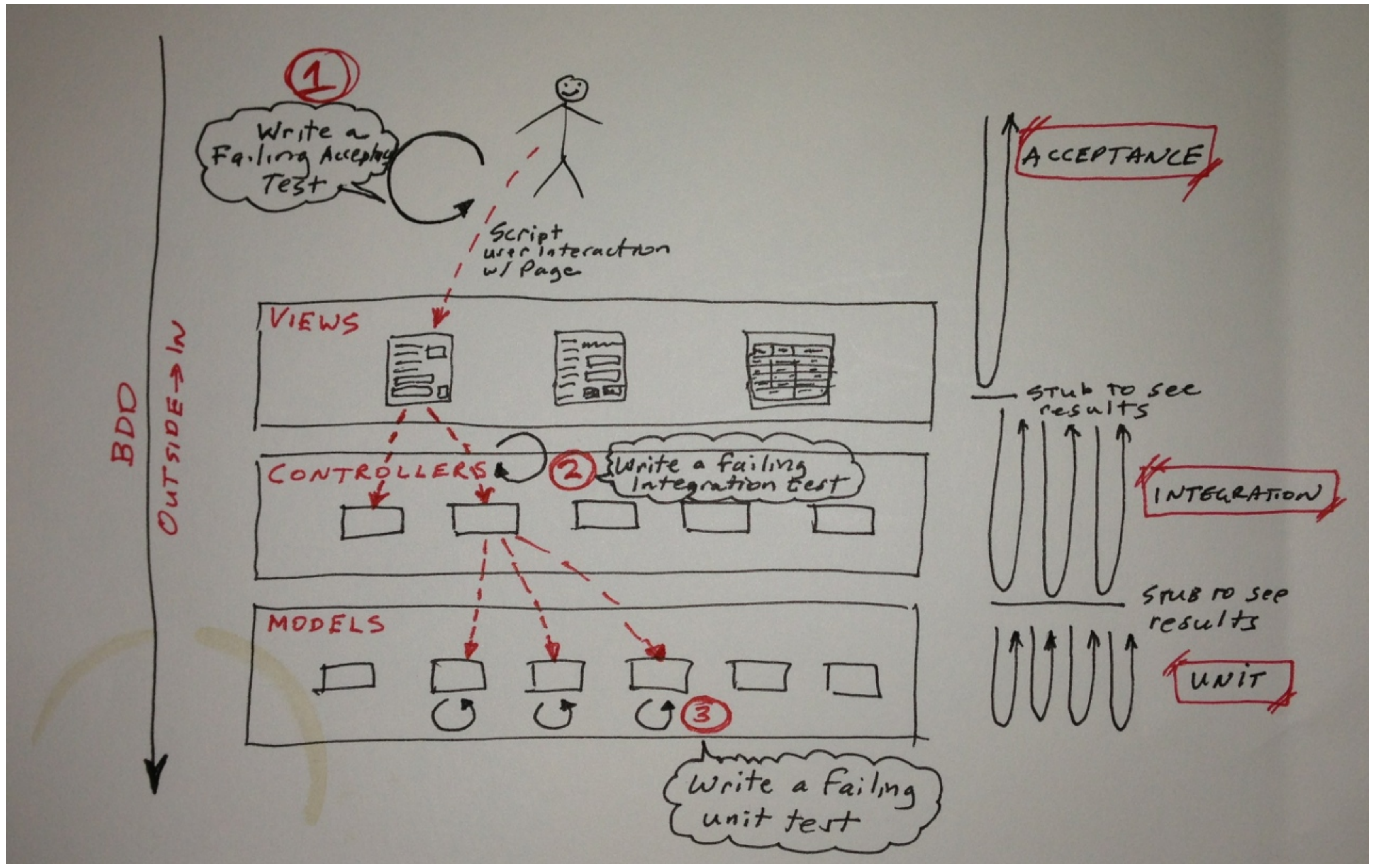
1. User requests /projects
2. Rails router forwards the request to projects_controller#index action
3. The index action creates the instance variable @projects by using the Project model all method
4. The all method is mapped by ActiveRecord to a select statement for your DB
5. @projects returns back with a collection of all Project objects
6. The index action renders the index.html.erb view
7. An HTML table of Projects is rendered using ERB (embedded Ruby) which has access to the @projects variable
8. The HTML response is returned to the User



Testing

A Means To An End

- Outside-in Testing, BDD/TDD, Unit, Integration and Acceptance testing in one picture:



1
Write a Failing Acceptance Test
Script user interaction w/ Page
VIEWS
CONTROLLERS
2
Write a failing Integration test
MODELS
3
Write a Failing unit test
BDD
OUTSIDE → IN
ACCEPTANCE
Stub to see results
INTEGRATION
Stub to see results
UNIT

BDD

Behavior-Driven Development

BDD
Behavior Driven Development

- BDD focuses TDD to deliver the maximum value possible to stakeholders
- BDD is a refinement in the language and tooling used for TDD
- As the name implies with BDD we focus on behavior specifications
- Typically BDD works from the outside in, that is starting with the parts of the software whose behavior is directly perceive by the user
- We say BDD refines TDD in that there is an implicit decoupling of the tests and the implementation (i.e.. don't tests implementation specifics, test perceived behavior)

BDD
Behavior Driven Development

- BDD focuses on “specifications” that describe the behavior of the system
- In the process of fleshing out a story the specifications start from the outside and might move towards the inside based on need
- In the context of a Web Application this Outside-In approach typically means that we are starting with specifications related to the User Interface
- If we are talking about a software component then we mean the API for said component

BDD
Behavior Driven Development

- **BDD** helps us figure out **what to test**, where to start and **what to ignore** (or what to make a target of opportunity)
  - **What** to test? ➜ Use Cases or User Stories, test what something **does (behavior)** rather than what something **is (structure)**
  - **Where** to start? ➜ From the outer most layer
  - **What** to ignore? ➜ Anything else… Until proven that you can’t

BDD
Behavior Driven Development

- BDD focuses on **getting the words right**, the resulting specifications become an **executable/self-verifying** form of **documentation**

- BDD specifications follow a format that makes them easy to be driven by your system's User Stories

*As a [**role**], I want [**goal**] so that [**benefit**]*

User Story

→

*Given [**role and its state**]*
*When [**an event/action occurs**]*
*Then [**the benefit**]*

Specification

TDD with RSpec

Mini-Tutorial

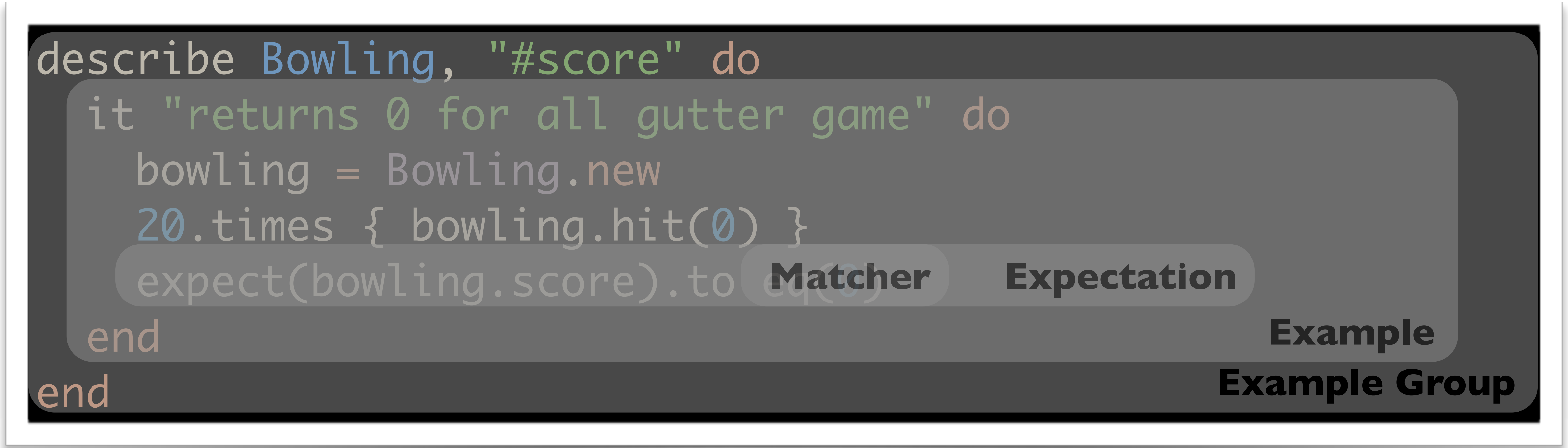
RSpec
Ruby's BDD Framework

- RSpec is the **most popular BDD** framework for Ruby
- Created by Steven Baker in 2005, enhanced and maintained by David Chelimsky until late 2012
- RSpec provides a DSL to write executable examples of the expected behavior of a piece of code in a controlled context

RSpec

Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods



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

RSpec
Matchers

- RSpec comes built in with a nice collection of matchers, including:

```
be_true  # passes if actual is truthy (not nil or false)
be_false # passes if actual is falsy (nil or false)
be_nil   # passes if actual is nil
be       # passes if actual is truthy (not nil or false)

expect { ... }.to raise_error
expect { ... }.to raise_error(ErrorClass)
expect { ... }.to raise_error("message")
expect { ... }.to raise_error(ErrorClass, "message")

expect { ... }.to throw_symbol
expect { ... }.to throw_symbol(:symbol)
expect { ... }.to throw_symbol(:symbol, 'value')

be_xxx         # passes if actual.xxx?
have_xxx(:arg) # passes if actual.has_xxx?(:arg)
```

RSpec
Matchers

- and ...

```
be_empty

be(expected) # passes if actual.equal?(expected)
eq(expected) # passes if actual == expected

== expected     # passes if actual == expected
eql(expected)   # passes if actual.eql?(expected)
equal(expected) # passes if actual.equal?(expected)

be >  expected
be >= expected
be <= expected
be <  expected
=~ /expression/
match(/expression/)
be_within(delta).of(expected)

be_instance_of(expected)
be_kind_of(expected)
```

https://www.relishapp.com/rspec/rspec-expectations/v/2-13/docs/built-in-matchers

Test-Driven Development

Drive your Development with Tests

- TDD is **not** *really* **about testing**
- TDD is a **design technique**
- TDD leads to **cleaner code** with **separation** of **concerns**
- Cleaner code is more reliable and easier to maintain (Duh)

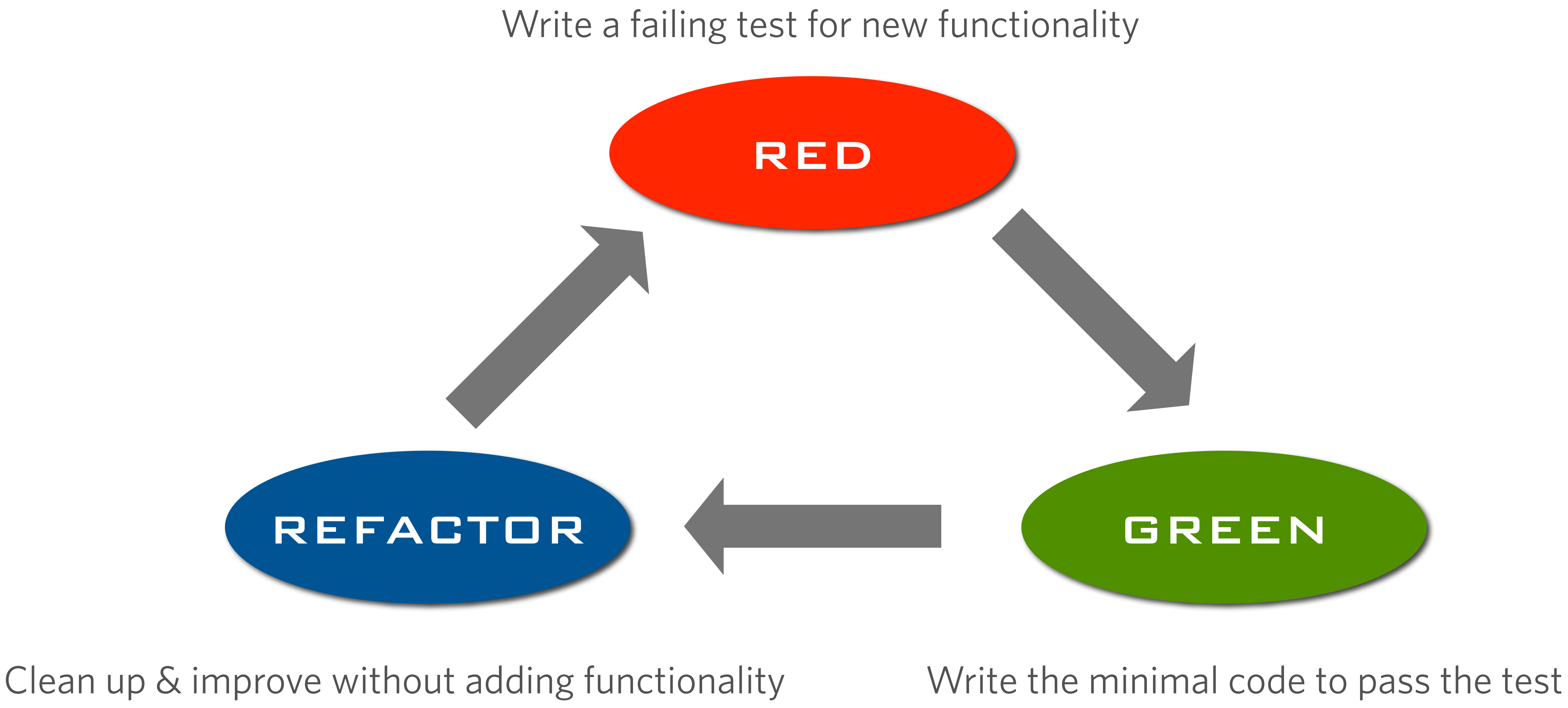
Test-Driven Development
Drive your Development with Tests

- TDD creates a **tight loop of development** that **cognitively engages us**
- TDD gives us **lightweight rigor** by making development, **goal-oriented** with a **clear goal setting**, goal reaching and improvement stages
- The stages of TDD are commonly known as the **Red-Green-Refactor** loop

Test-Driven Development

Drive your Development with Tests

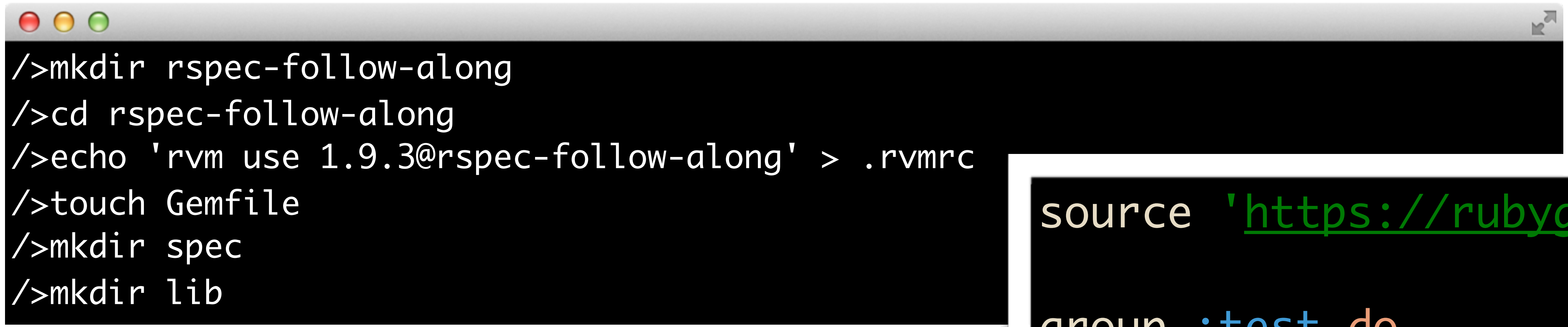
- The **Red-Green-Refactor** Loop:


Write a failing test for new functionality
RED
GREEN
REFACTOR
Clean up & improve without adding functionality
Write the minimal code to pass the test

Test-Driven Development

Evolve your Code with Tests

- Let's work through **a simple TDD/BDD exercise** using RSpec
- We'll design a **simple shopping cart** class
- We'll start by creating a new folder for our exercise and adding a **.rvmrc** file and a **Gemfile**



```
/>mkdir rspec-follow-along
/>cd rspec-follow-along
/>echo 'rvm use 1.9.3@rspec-follow-along' > .rvmrc
/>touch Gemfile
/>mkdir spec
/>mkdir lib
```

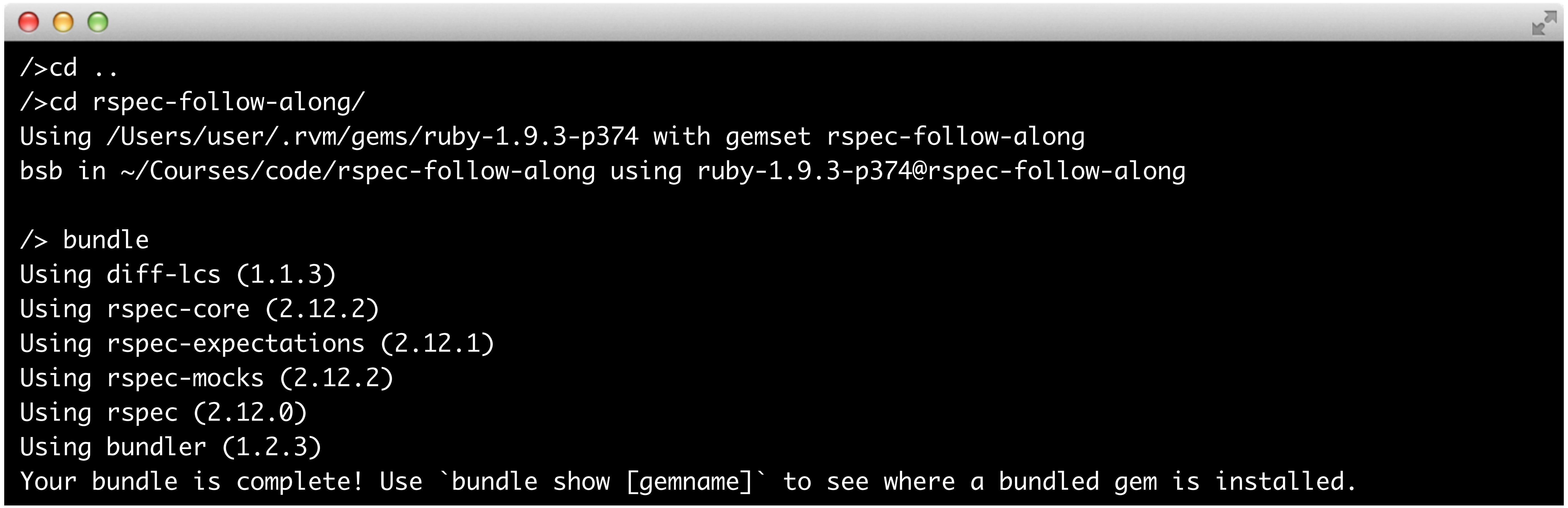
```
source 'https://rubygems.org'

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

Test-Driven Development

Evolve your Code with Tests

FOLLOW ALONG

- With our project configure for RVM and a Gemfile in place we can reenter the directory to activate the Gemset and run the bundle command:



```
/>cd ..
/>cd rspec-follow-along/
Using /Users/user/.rvm/gems/ruby-1.9.3-p374 with gemset rspec-follow-along
bsb in ~/Courses/code/rspec-follow-along using ruby-1.9.3-p374@rspec-follow-along

/> bundle
Using diff-lcs (1.1.3)
Using rspec-core (2.12.2)
Using rspec-expectations (2.12.1)
Using rspec-mocks (2.12.2)
Using rspec (2.12.0)
Using bundler (1.2.3)
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem is installed.
```

Test-Driven Development

Evolve your Code with Tests

Follow Along

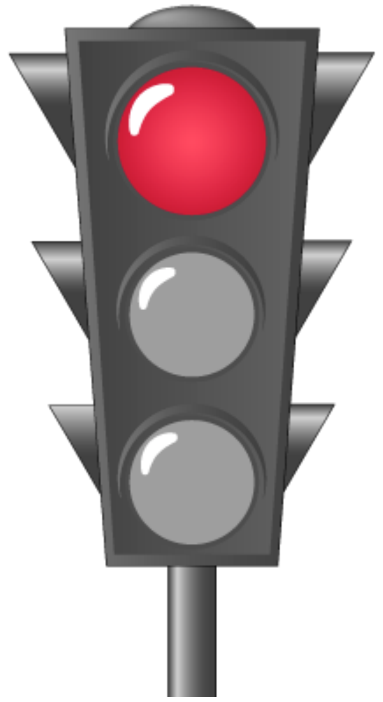
- We'll start the RGR loop with the simplest possible failure:
  "There is no Cart!"
- Create the file **cart_spec.rb** in the spec directory with the following contents:

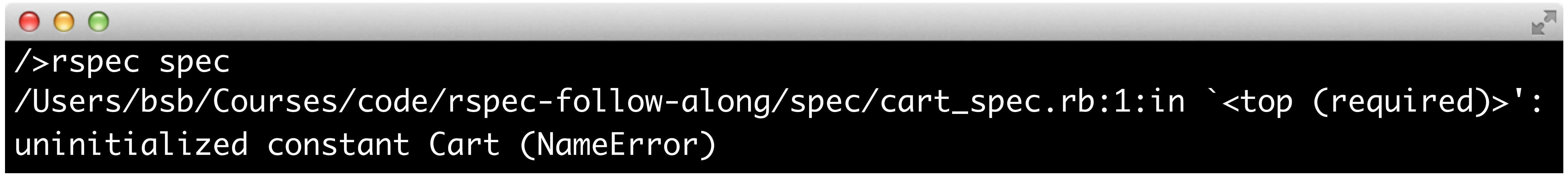
```
describe Cart do

end
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- Let's run the specs using the **rspec command** and passing the spec directory as an argument
- Have we arrived at the **RED** state in our red-green-refactor cycle?





```
/>rspec spec
/Users/bsb/Courses/code/rspec-follow-along/spec/cart_spec.rb:1:in `<top (required)>':
uninitialized constant Cart (NameError)
```

Hint: if you have a failure with no tests it typically means that you need a test (but let's ignore that for a second...)

Test-Driven Development
Evolve your Code with Tests

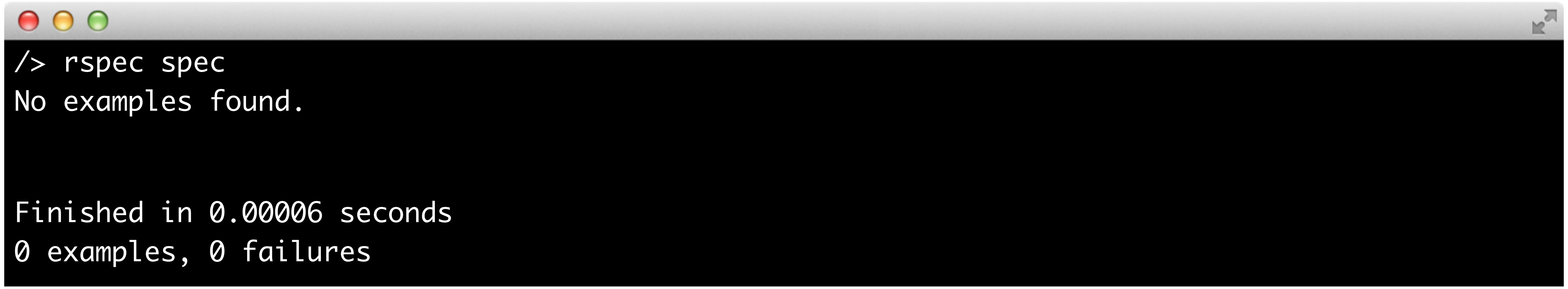
FOLLOW ALONG

- Let's create the **Cart class in a /lib** folder and require it in our spec:

```ruby
class Cart
end
```

```ruby
require_relative '../lib/cart.rb'

describe Cart do
end
```

- Now we have no failures but also we have no specs...

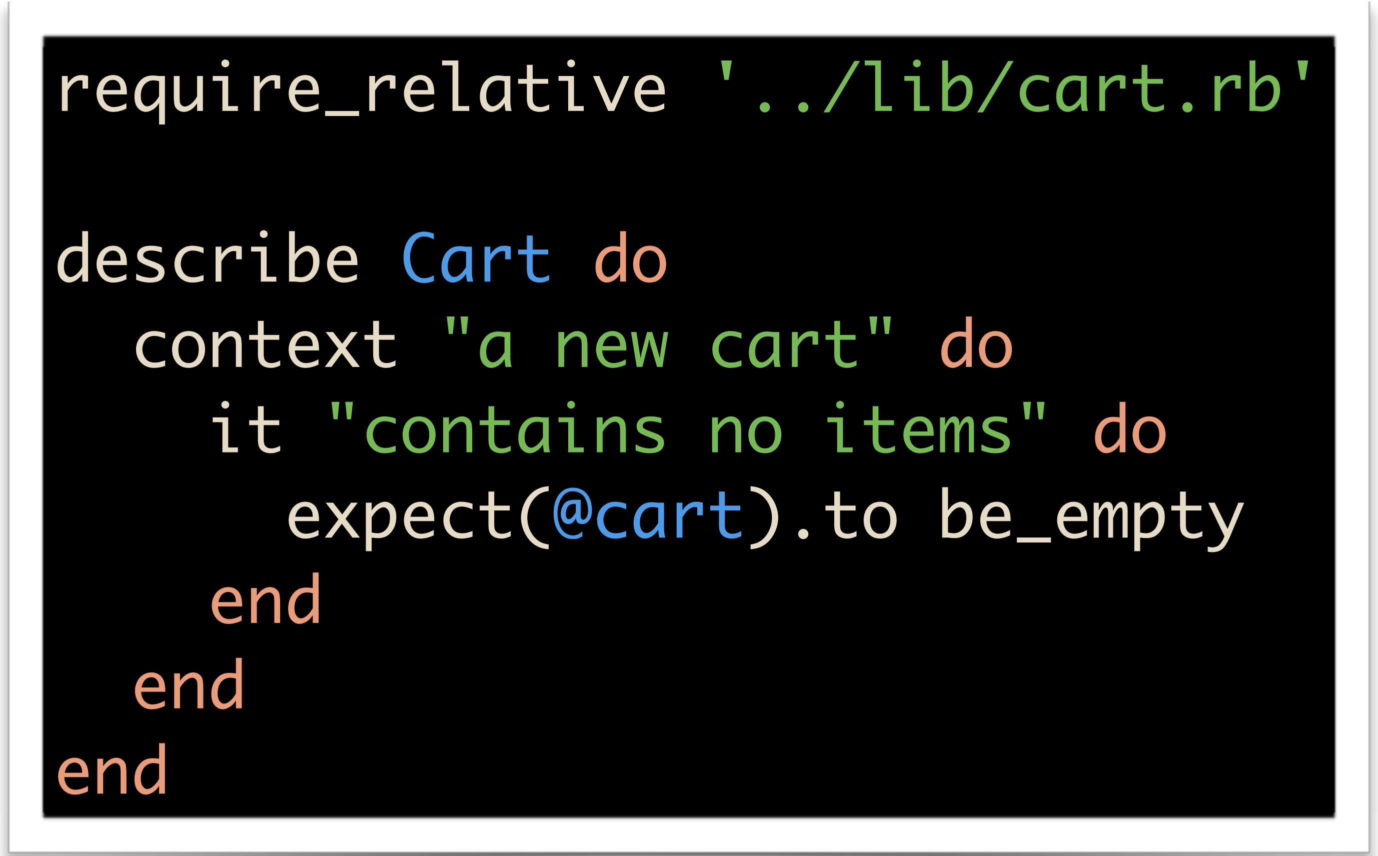


```
/> rspec spec
No examples found.


Finished in 0.00006 seconds
0 examples, 0 failures
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- Let’s craft our first real test to drive the development of the Cart
- The spec to tackle is: ***“An instance of Cart when new contains no items”***



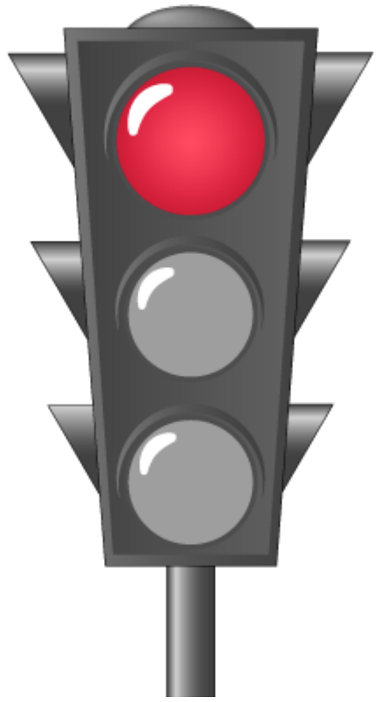
```ruby
require_relative '../lib/cart.rb'

describe Cart do
  context "a new cart" do
    it "contains no items" do
      expect(@cart).to be_empty
    end
  end
end
```

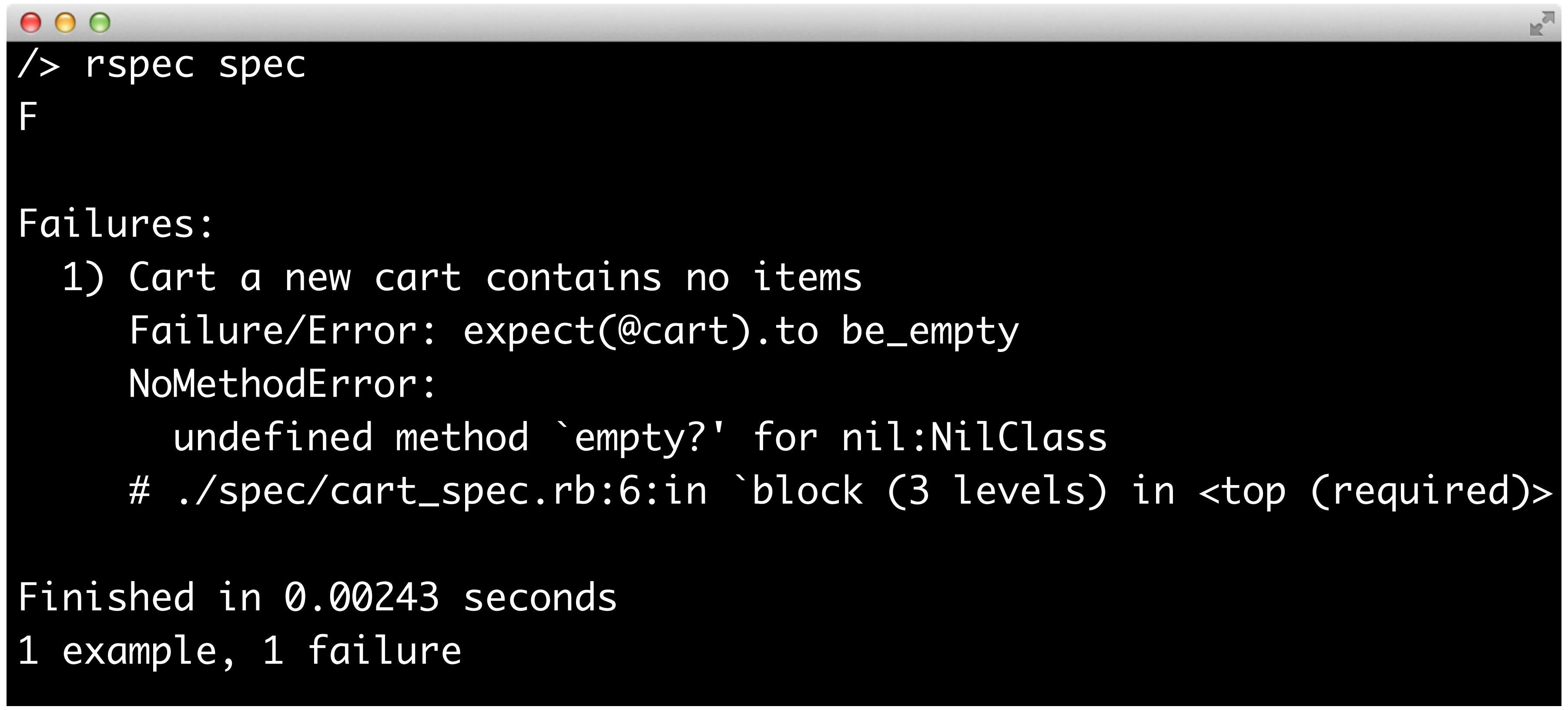
Test-Driven Development

Evolve your Code with Tests

Follow Along

- If we run the specs we can see a failure:





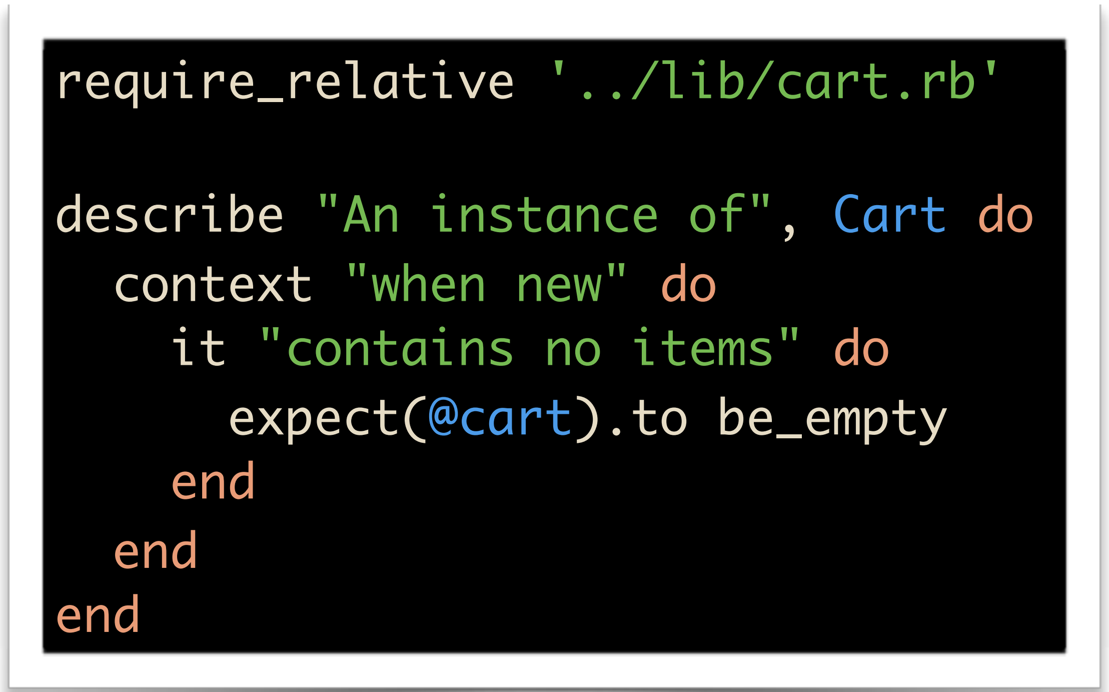
```
/> rspec spec
F

Failures:
  1) Cart a new cart contains no items
     Failure/Error: expect(@cart).to be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:6:in `block (3 levels) in <top (required)>

Finished in 0.00243 seconds
1 example, 1 failure
```

Now we have our first "real" test-driven failure (and that is a good thing!)

Test-Driven Development
Evolve your Code with Tests

FOLLOW ALONG

- One of the mantras of BDD is to ***"get the words right"***
- If you noticed on the last run the spec output read as "*Cart a new cart contains no items*"
- RSpec is flexible enough to allow us to pass a string to be prefixed to the describe block to make tailor the output to our needs

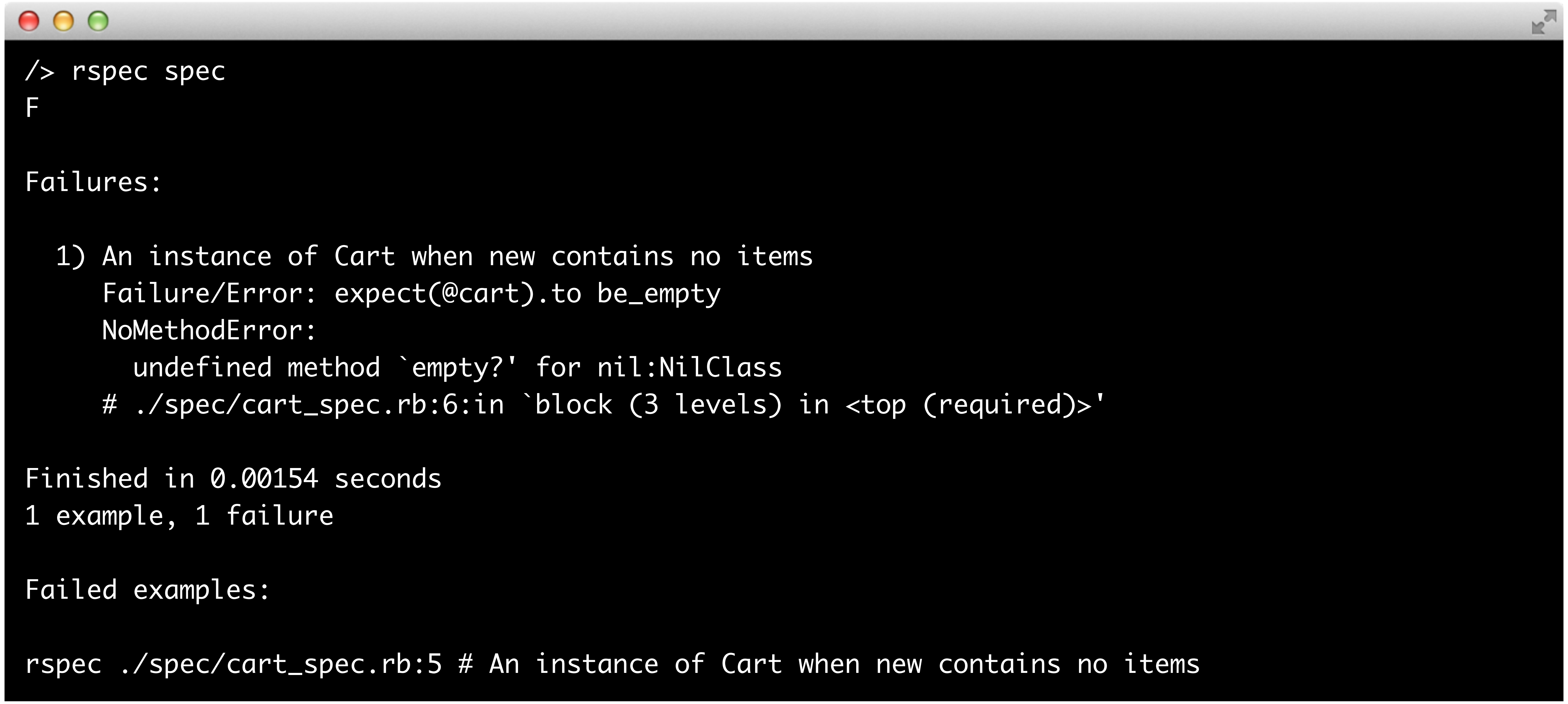


```ruby
require_relative '../lib/cart.rb'

describe "An instance of", Cart do
  context "when new" do
    it "contains no items" do
      expect(@cart).to be_empty
    end
  end
end
```

Test-Driven Development
Evolve your Code with Tests

FOLLOW ALONG

- If we run the specs we can see that the output now matches the desire spec wording

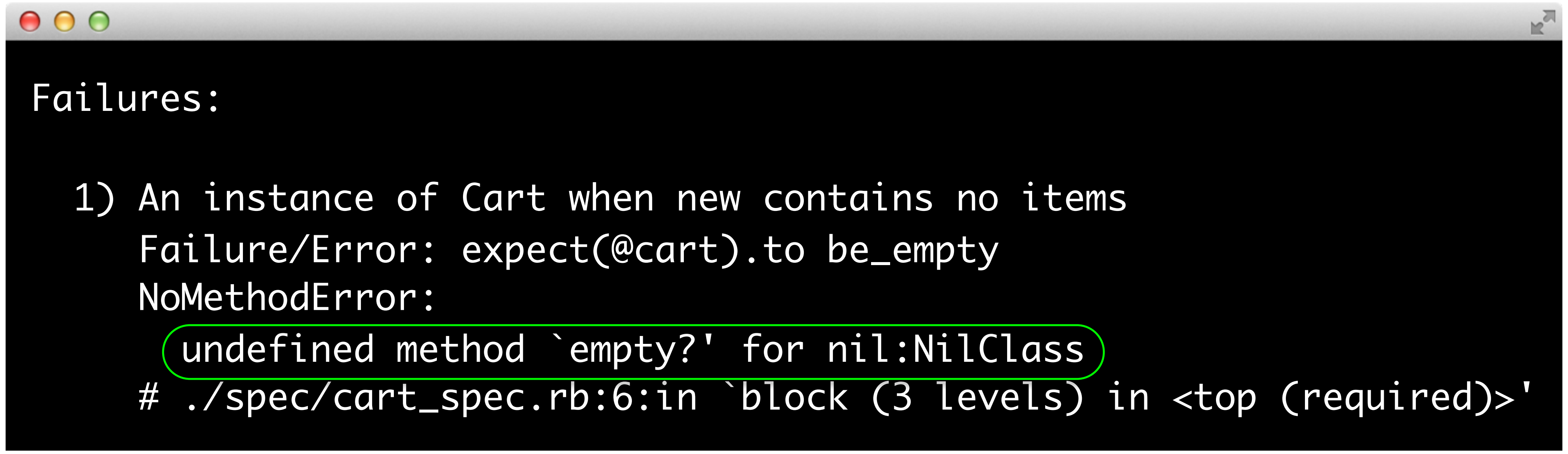


```
/> rspec spec
F

Failures:

  1) An instance of Cart when new contains no items
     Failure/Error: expect(@cart).to be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:6:in `block (3 levels) in <top (required)>'

Finished in 0.00154 seconds
1 example, 1 failure

Failed examples:

rspec ./spec/cart_spec.rb:5 # An instance of Cart when new contains no items
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- The output shows that **we have two failures**, one implicit and one explicit
- **Explicit Failure:** We are assuming that a cart has an **#empty?** method
- **Implicit Failure:** The instance variable **@cart** has not been initialized



```
Failures:

  1) An instance of Cart when new contains no items
     Failure/Error: expect(@cart).to be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:6:in `block (3 levels) in <top (required)>'
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- We'll start by addressing the fact that our test fixture hasn't been setup
- Just adding the line `@cart = Cart.new` wouldn't be very TDDish
- What we should do is first **make the failure explicit** by writing a test for it!

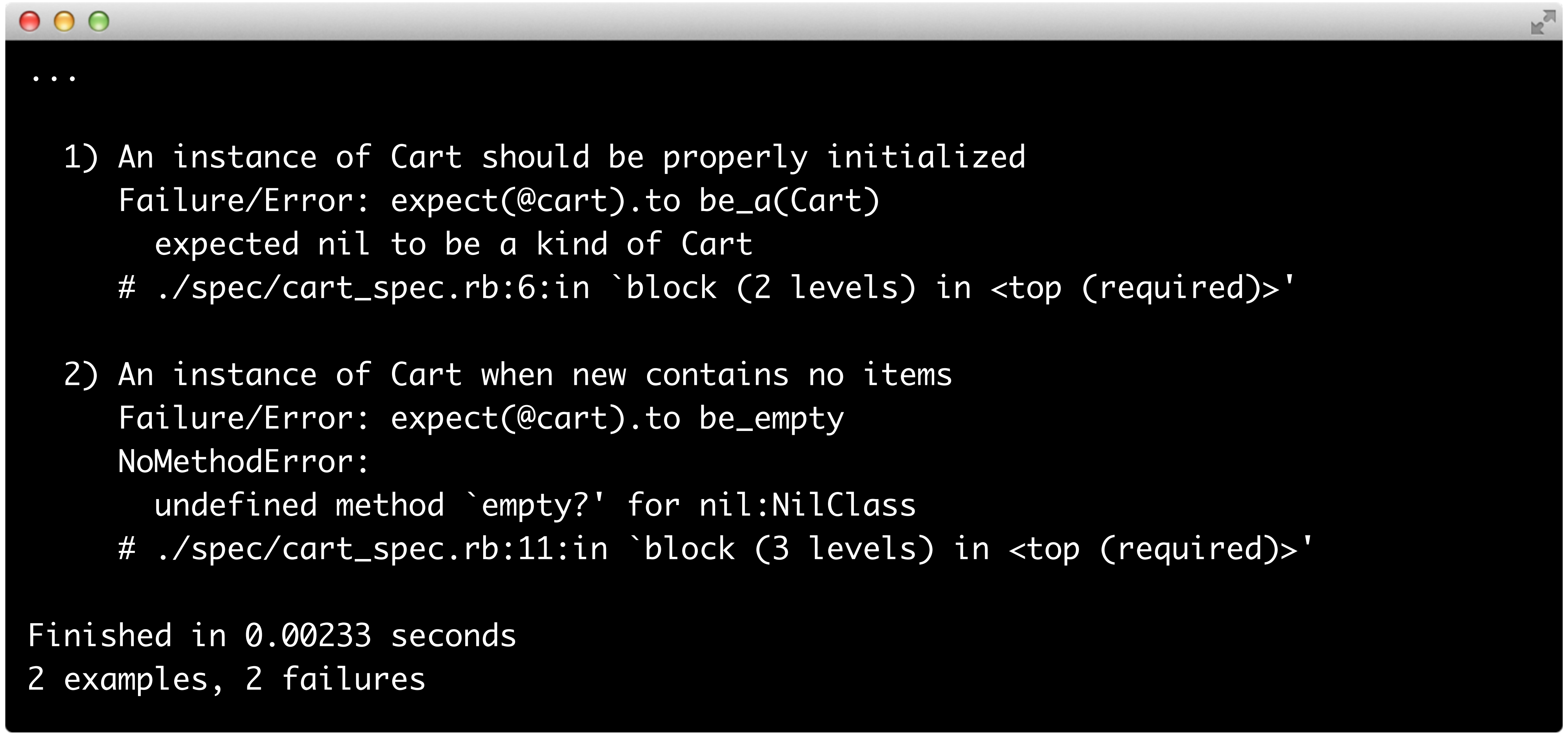
```
require_relative '../lib/cart.rb'

describe "An instance of", Cart do

  it "should be properly initialized" do
    expect(@cart).to be_a(Cart)
  end

...
```

Remember our initial failure with no tests?

Test-Driven Development
Evolve your Code with Tests

FOLLOW ALONG

- Now we have two valid failing tests to pass, let's get on with it!



```
...

  1) An instance of Cart should be properly initialized
     Failure/Error: expect(@cart).to be_a(Cart)
       expected nil to be a kind of Cart
     # ./spec/cart_spec.rb:6:in `block (2 levels) in <top (required)>'

  2) An instance of Cart when new contains no items
     Failure/Error: expect(@cart).to be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:11:in `block (3 levels) in <top (required)>'

Finished in 0.00233 seconds
2 examples, 2 failures
```

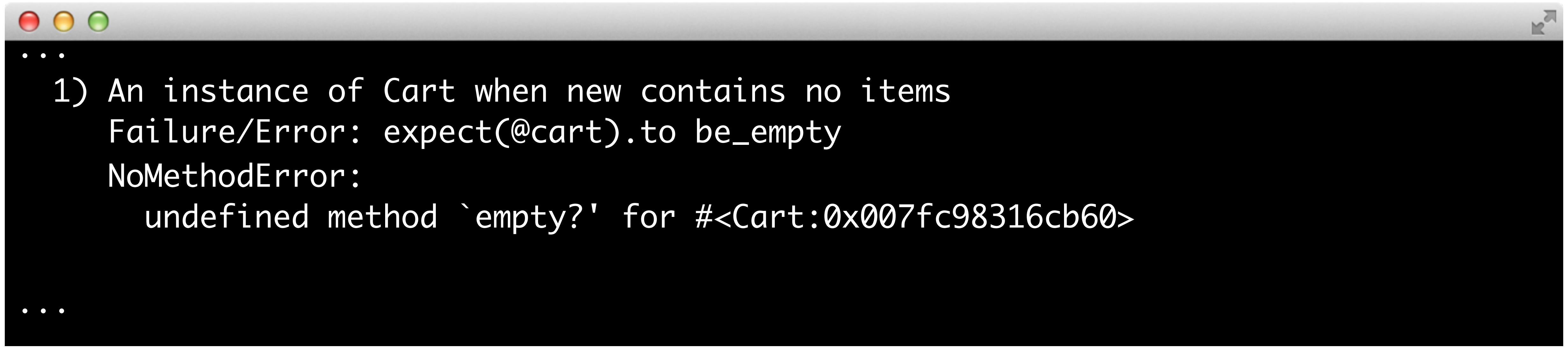
Test-Driven Development

Evolve your Code with Tests

Follow Along

- We'll pass the test by adding the line `@cart = Cart.new` in a before-each block:

```ruby
describe "An instance of", Cart do

  before :each do
    @cart = Cart.new
  end
```



```
...
  1) An instance of Cart when new contains no items
     Failure/Error: expect(@cart).to be_empty
     NoMethodError:
       undefined method `empty?' for #<Cart:0x007fc98316cb60>


...
```

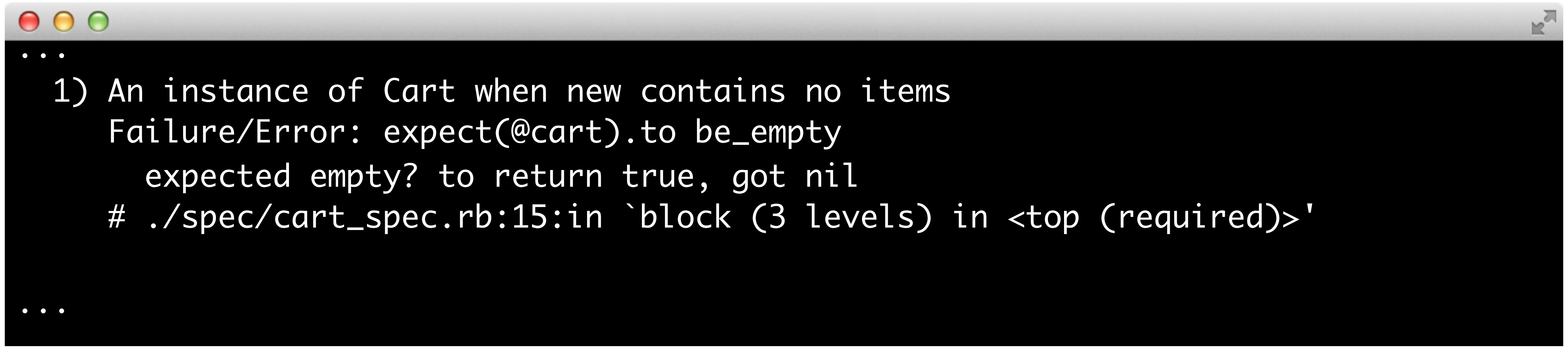
Test-Driven Development

Evolve your Code with Tests

Follow Along

- Let's add a skeleton empty? method to the Cart class:

```ruby
class Cart
  def empty?
    nil
  end
end
```



```
...
  1) An instance of Cart when new contains no items
     Failure/Error: expect(@cart).to be_empty
       expected empty? to return true, got nil
     # ./spec/cart_spec.rb:15:in `block (3 levels) in <top (required)>'


...
```

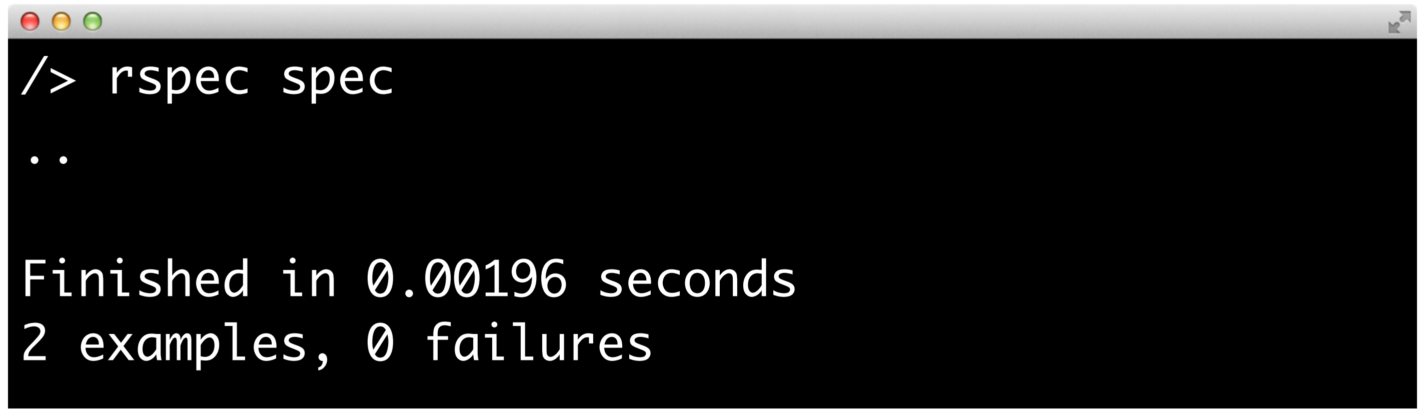
Test-Driven Development

Evolve your Code with Tests

FOLLOW ALONG

- Now we can comply with the spec by providing an implementation of our Cart
- In this case we are using a Hash to hold our items and delegating to the @items#empty? method

```ruby
class Cart

  def initialize
    @items = {}
  end

  def empty?
    @items.empty?
  end
end
```



```
/> rspec spec
..

Finished in 0.00196 seconds
2 examples, 0 failures
```

We've reached the GREEN state

Test-Driven Development

Drive your Development with Tests

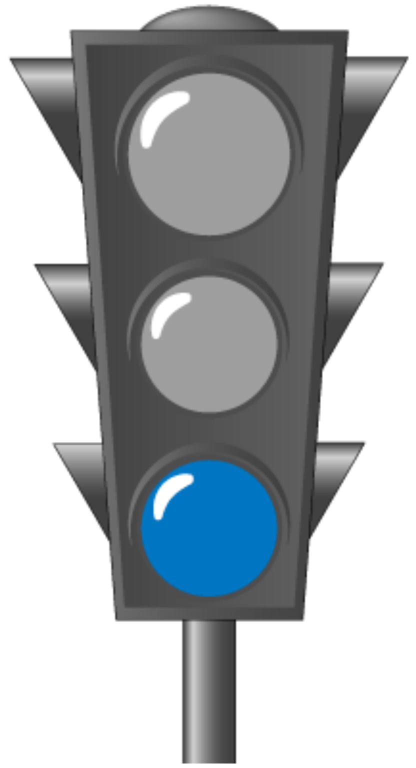
- In the **REFACTOR** state we concentrate on **making the current implementation better**, cleaner and more robust
- It is very likely that **early** on in the development **there won't be much to refactor**
- The need for **refactoring is a side-effect of increasing complexity** and interaction between classes and subsystems
- Refactoring can **also introduce implementation specific specs** or **reveal holes in your previous specs**

Test-Driven Development

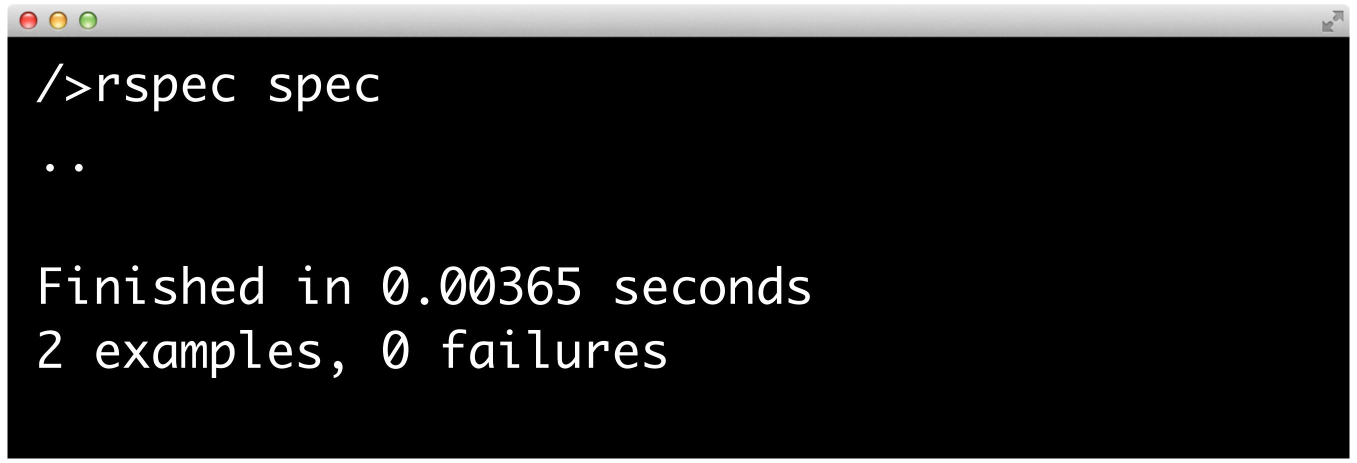
Evolve your Code with Tests

FOLLOW ALONG

- Let's use Ruby's **Forwardable** module to simplify the delegation of the collection methods to the @items Hash:



```ruby
class Cart
  extend Forwardable
  def_delegator :@items, :empty?
  def initialize
    @items = {}
  end
end
```

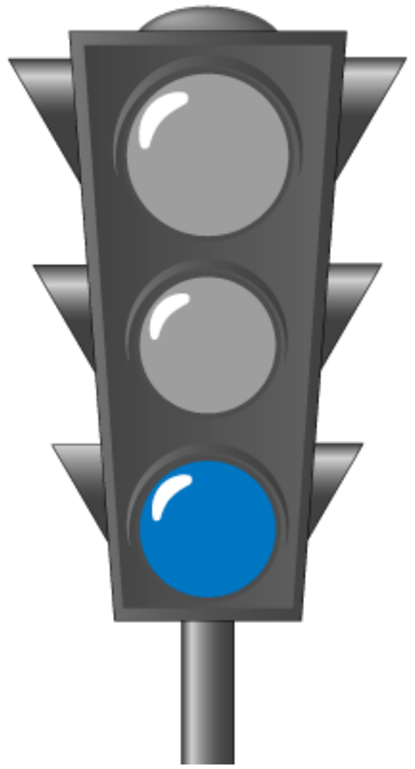


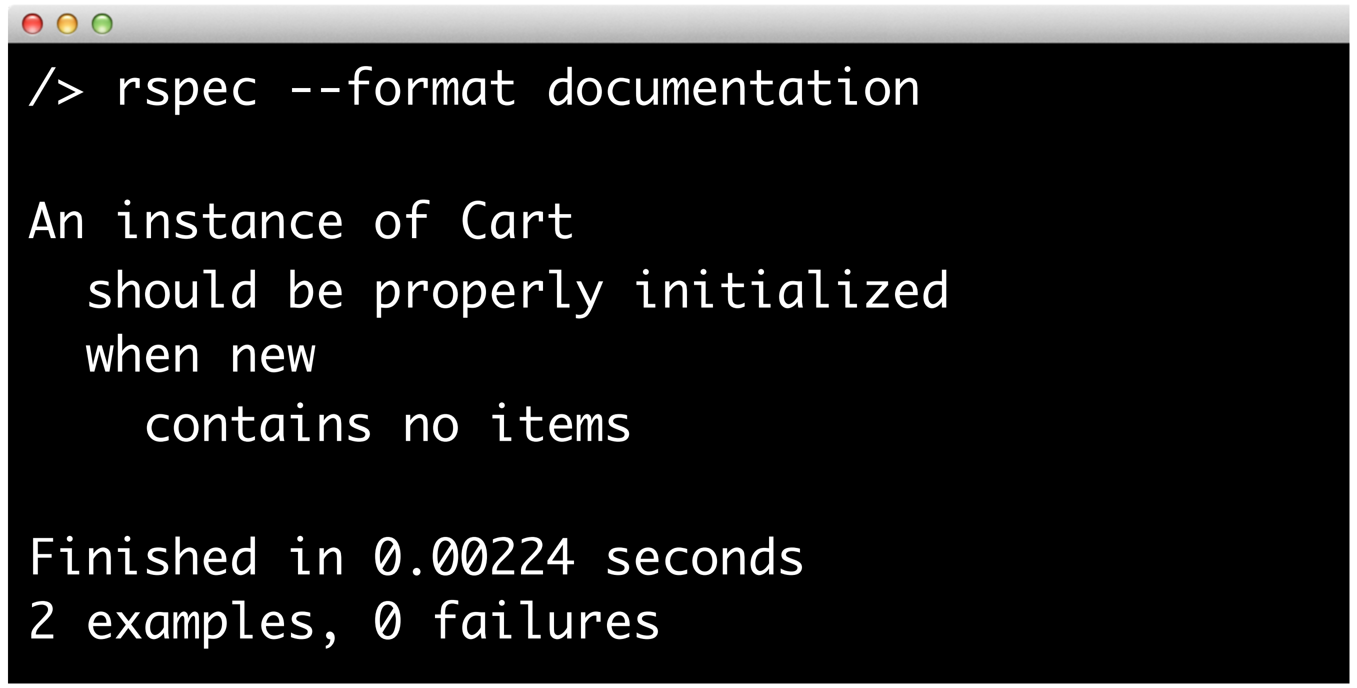
```
/>rspec spec
..

Finished in 0.00365 seconds
2 examples, 0 failures
```

Test-Driven Development
RSpec Command

Follow Along

- The RSpec command provides several arguments to tailor the run and output of your specifications
- For example to see the group and example names in the output use **--format documentation**





```
/> rspec --format documentation

An instance of Cart
  should be properly initialized
  when new
    contains no items

Finished in 0.00224 seconds
2 examples, 0 failures
```

Lab 1.0
TDD & BDD

- Lab 1.0 consists of 4 specs to be implemented in a TDD fashion:
  - ***"An new and empty cart total value should be $0.0"***
  - ***"An empty cart should no longer be empty after adding an item"***
  - ***"A cart with items should have a total value equal to the sum of each items' value times its quantity"***
  - ***"Increasing the quantity of an item should not increase the Carts' unique items count"***

BDD with Capybara

Mini-Tutorial

Acceptance
Acceptance Testing

- Proper **acceptance tests** treat your **application** as a **black box**
- They should **know as little as possible** about what happens under the hood
- They're **just there to interact** with the interface and **observe the results**
- Jeff Casimir says...

  "A great testing strategy is to extensively cover the data layer with unit tests then skip all the way up to acceptance tests. This approach gives great code coverage and builds a test suite that can flex with a changing codebase."

Capybara
Acceptance test framework for web applications

- Capybara is a **lightweight alternative to Cucumber**
- Capybara is a **browser automation** library
- Helps you test web applications by **simulating** how **a real user** would interact with your app
  - It is **agnostic about** the **driver** running your tests and comes with Rack::Test and Selenium support built in
  - **WebKit** is **supported through** an external **gem**

RSpec, Capybara, and Steak

Acceptance test framework for web applications

- Many developers **don’t want to bother with Cucumber**
- They want **outside-in testing without the translation step**
- The **Steak** project **integrated RSpec** and **Capybara** directly
- Now we can **write acceptance tests** just **like** you write **unit tests**, greatly simplifying the process
- In late 2010 the **Capybara absorbed the Steak syntax**

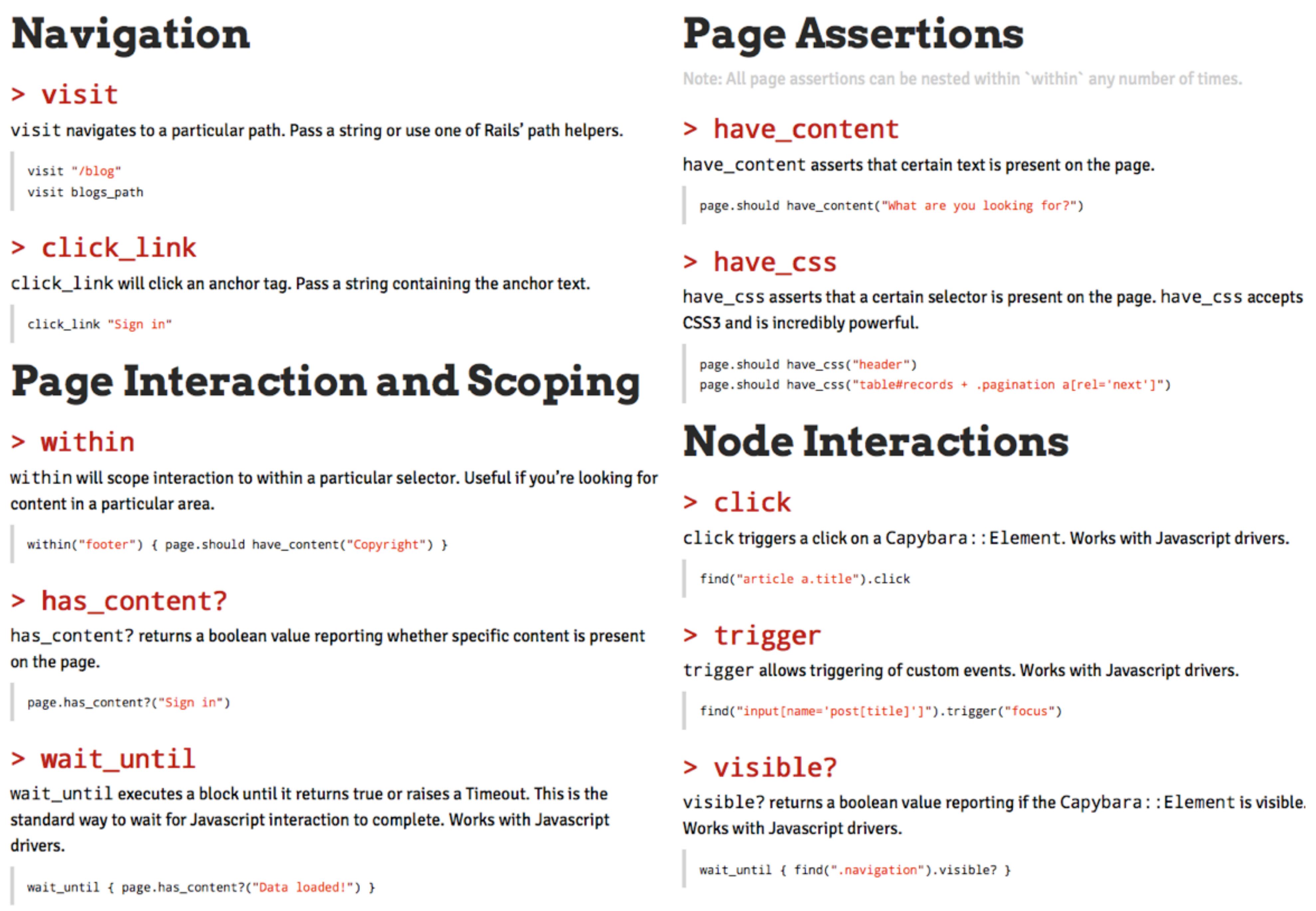
Rack::Test

Acceptance test framework for web applications

- Capybara uses **Rack::Test by default**
- **Rack::Test** interacts with your app **at** the **Rack level**
- It **runs requests** against your app, then **provides the resulting HTML** to **Capybara and RSpec** for examination.
- **Rack::Test is** completely **headless** (and therefore fast)
- The disadvantage is that **it doesn't process JavaScript** (or give you visual feedback)
- To test JavaScript in your acceptance tests you can use the **selenium-webdriver** or **capybara-webkit** driver.

Capybara DSL

How to Drive the Browser



Navigation

> visit

visit navigates to a particular path. Pass a string or use one of Rails' path helpers.

```
visit "/blog"
visit blogs_path
```

> click_link

click_link will click an anchor tag. Pass a string containing the anchor text.

```
click_link "Sign in"
```

Page Interaction and Scoping

> within

within will scope interaction to within a particular selector. Useful if you're looking for content in a particular area.

```
within("footer") { page.should have_content("Copyright") }
```

> has_content?

has_content? returns a boolean value reporting whether specific content is present on the page.

```
page.has_content?("Sign in")
```

> wait_until

wait_until executes a block until it returns true or raises a Timeout. This is the standard way to wait for Javascript interaction to complete. Works with Javascript drivers.

```
wait_until { page.has_content?("Data loaded!") }
```

Page Assertions

Note: All page assertions can be nested within `within` any number of times.

> have_content

have_content asserts that certain text is present on the page.

```
page.should have_content("What are you looking for?")
```

> have_css

have_css asserts that a certain selector is present on the page. have_css accepts CSS3 and is incredibly powerful.

```
page.should have_css("header")
page.should have_css("table#records + .pagination a[rel='next']")
```

Node Interactions

> click

click triggers a click on a Capybara::Element. Works with Javascript drivers.

```
find("article a.title").click
```

> trigger

trigger allows triggering of custom events. Works with Javascript drivers.

```
find("input[name='post[title]']").trigger("focus")
```

> visible?

visible? returns a boolean value reporting if the Capybara::Element is visible. Works with Javascript drivers.

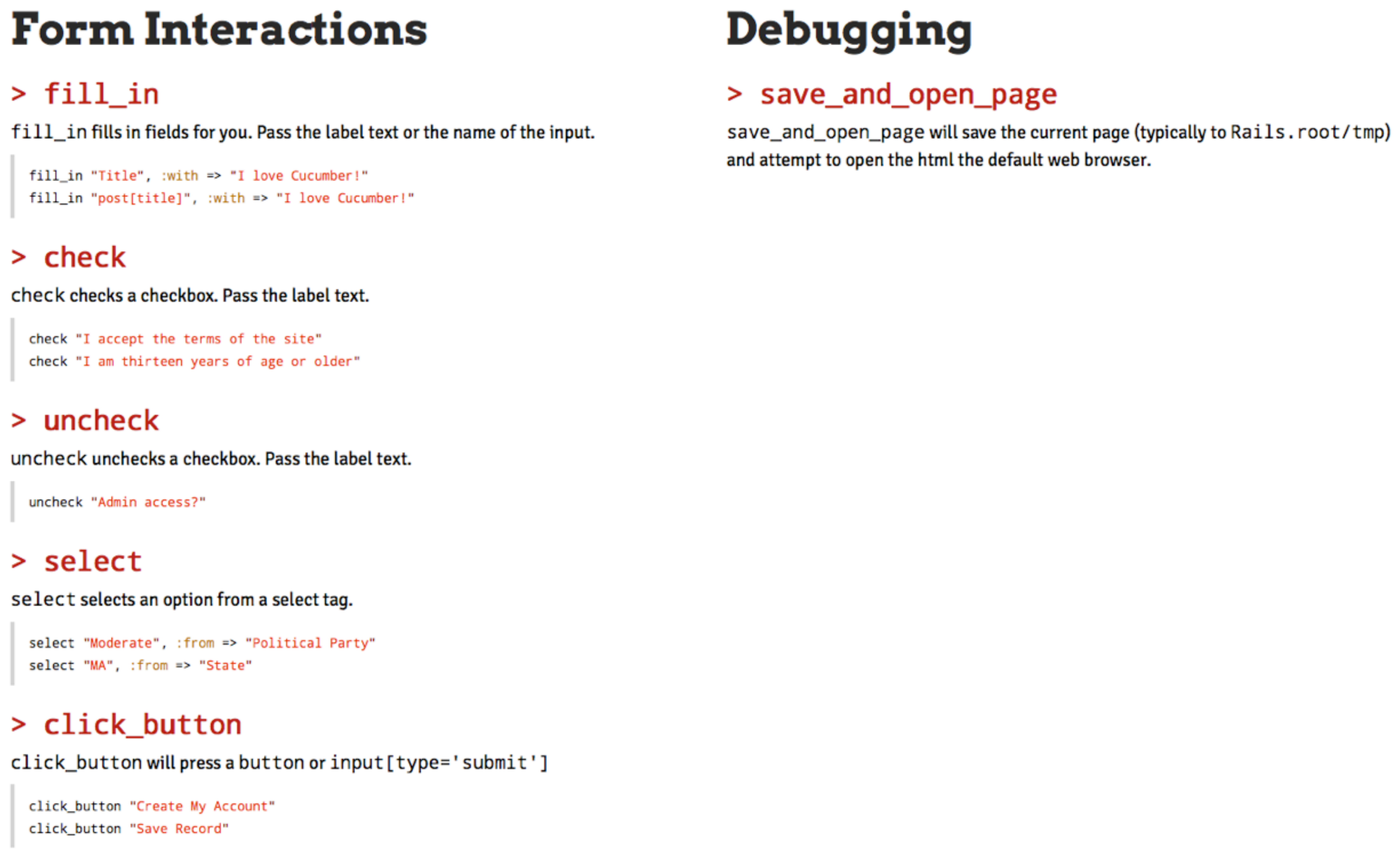
```
wait_until { find(".navigation").visible? }
```

- The folks at **ThoughtBot** put a nice Capybara **Cheat Sheet**

https://learn.thoughtbot.com/test-driven-rails-resources/capybara.pdf

Capybara DSL

How to Drive the Browser



Form Interactions

> fill_in

fill_in fills in fields for you. Pass the label text or the name of the input.

```
fill_in "Title", :with => "I love Cucumber!"
fill_in "post[title]", :with => "I love Cucumber!"
```

> check

check checks a checkbox. Pass the label text.

```
check "I accept the terms of the site"
check "I am thirteen years of age or older"
```

> uncheck

uncheck unchecks a checkbox. Pass the label text.

```
uncheck "Admin access?"
```

> select

select selects an option from a select tag.

```
select "Moderate", :from => "Political Party"
select "MA", :from => "State"
```

> click_button

click_button will press a button or input[type='submit']

```
click_button "Create My Account"
click_button "Save Record"
```

Debugging

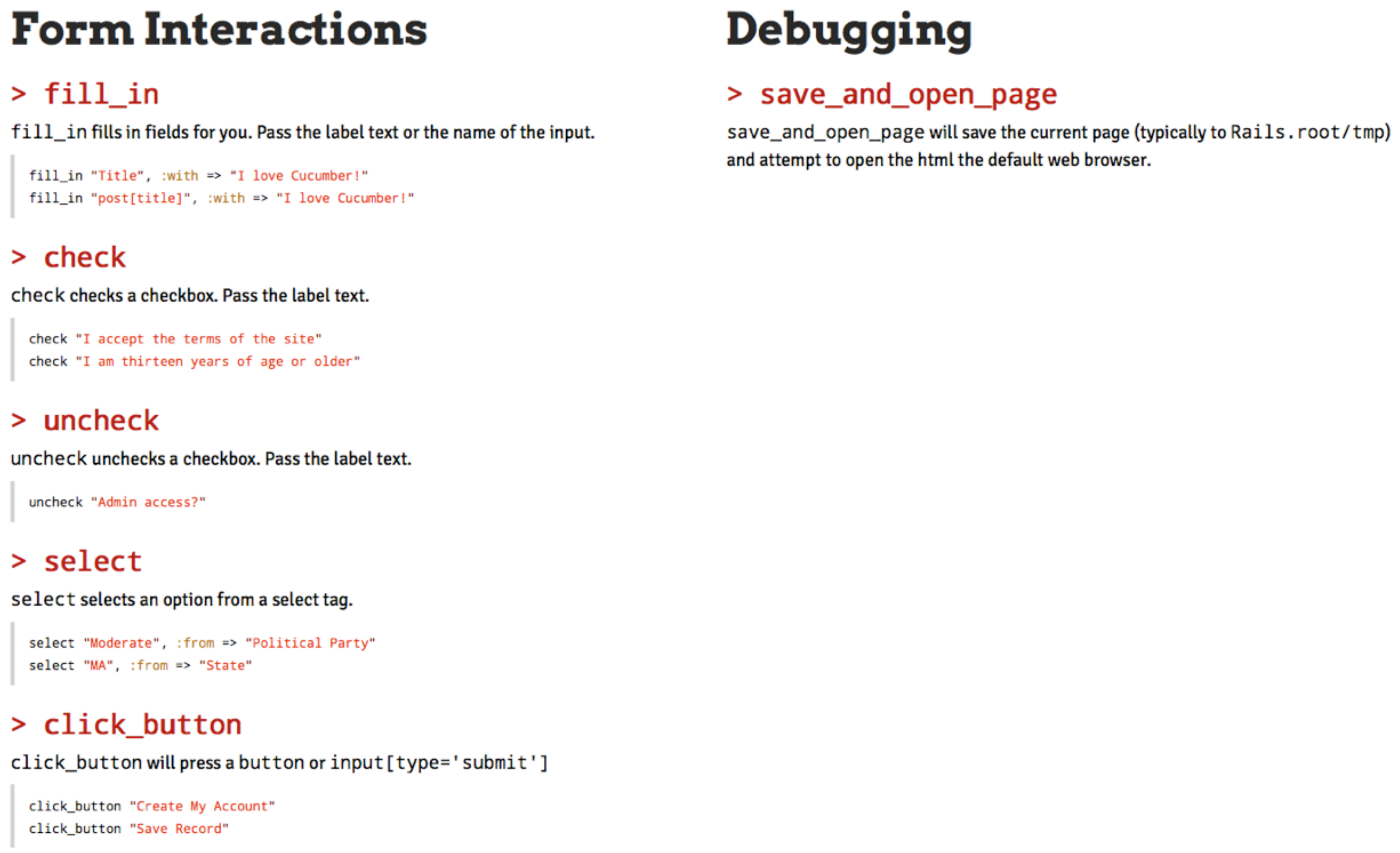
> save_and_open_page

save_and_open_page will save the current page (typically to Rails.root/tmp) and attempt to open the html the default web browser.

https://learn.thoughtbot.com/test-driven-rails-resources/capybara.pdf

Capybara DSL

How to Drive the Browser



Form Interactions

> fill_in

fill_in fills in fields for you. Pass the label text or the name of the input.

```
fill_in "Title", :with => "I love Cucumber!"
fill_in "post[title]", :with => "I love Cucumber!"
```

> check

check checks a checkbox. Pass the label text.

```
check "I accept the terms of the site"
check "I am thirteen years of age or older"
```

> uncheck

uncheck unchecks a checkbox. Pass the label text.

```
uncheck "Admin access?"
```

> select

select selects an option from a select tag.

```
select "Moderate", :from => "Political Party"
select "MA", :from => "State"
```

> click_button

click_button will press a button or input[type='submit']

```
click_button "Create My Account"
click_button "Save Record"
```

Debugging

> save_and_open_page

save_and_open_page will save the current page (typically to Rails.root/tmp) and attempt to open the html the default web browser.

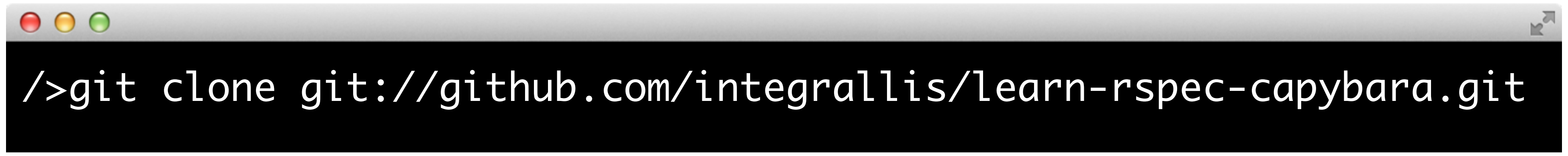
https://learn.thoughtbot.com/test-driven-rails-resources/capybara.pdf

Acceptance Testing w/ Capybara

Testing a Rails Application

FOLLOW ALONG

- We are going to mimic a user's interaction with a very simple Rails app
- The bulk of the functionality in the app will be provided by the **devise (authentication)** library and the **high_voltage (static pages)** gems
- To concentrate on the subtleties of Capybara we will test an existing application
- Let start by using git to clone the master branch of the repository at **https://github.com/integrallis/learn-rspec-capybara**



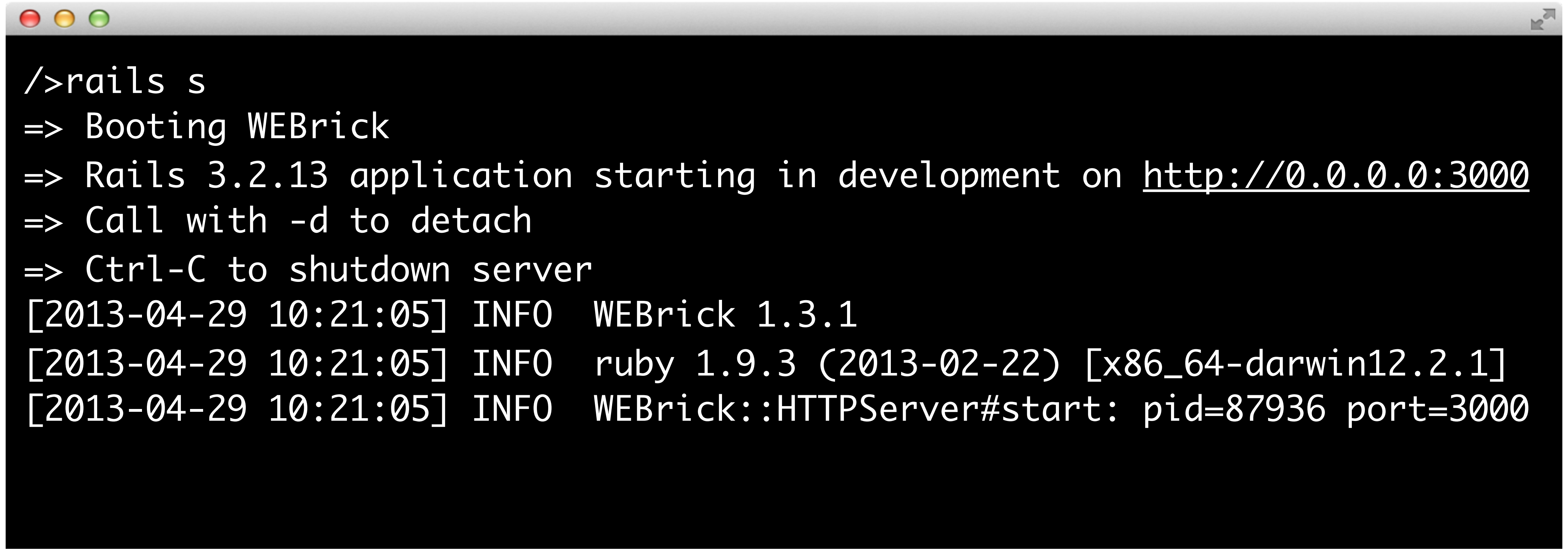
```
/>git clone git://github.com/integrallis/learn-rspec-capybara.git
```

Acceptance Testing w/ Capybara

Testing a Rails Application

FOLLOW ALONG

- CD in and out of the application to activate the RVM Gemset
- Bundle the application (bundle install), migrate it (rake db:migrate), prepare the test database (rake db:test:prepare) and launch it (rails s)

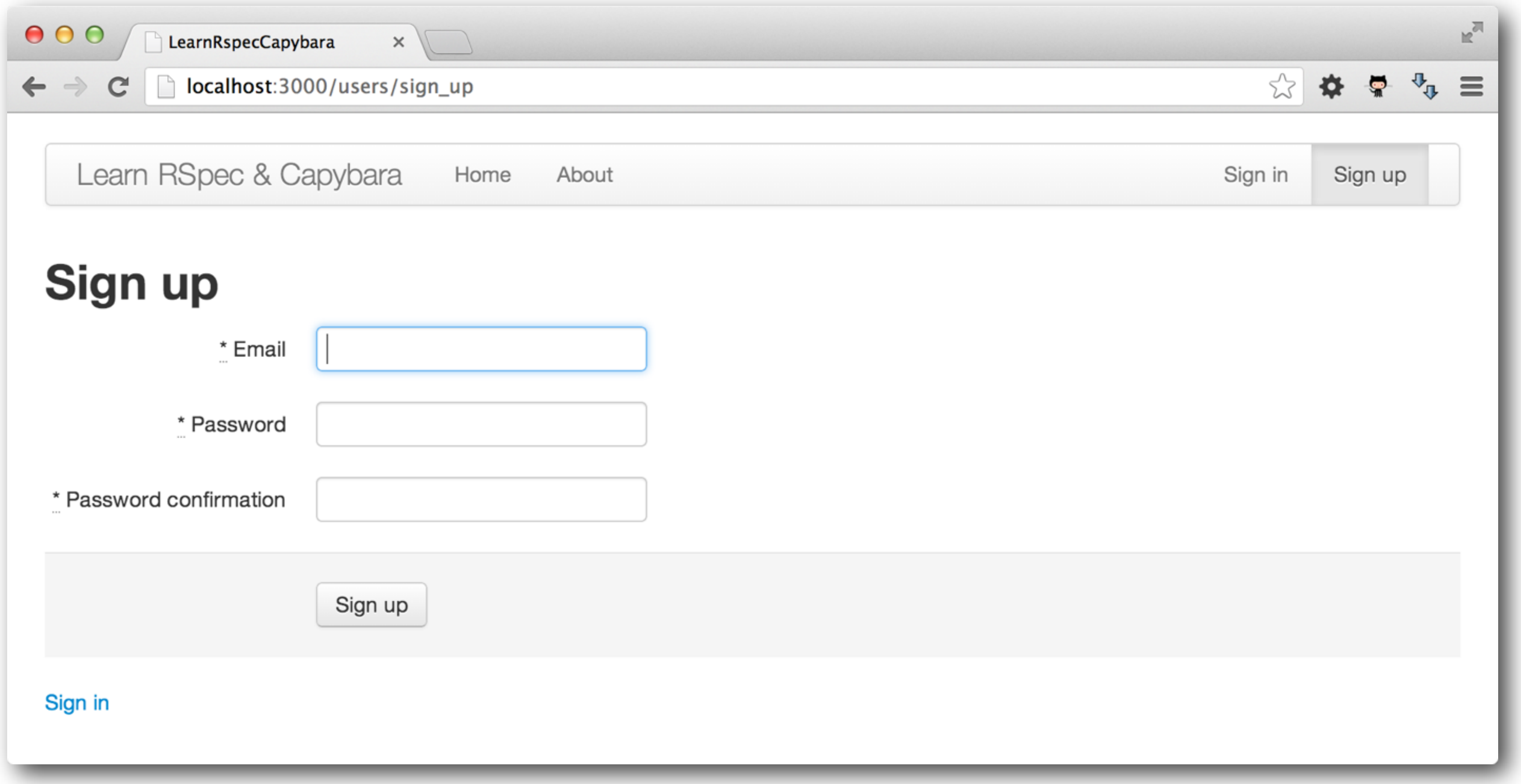


```
/>rails s
=> Booting WEBrick
=> Rails 3.2.13 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2013-04-29 10:21:05] INFO  WEBrick 1.3.1
[2013-04-29 10:21:05] INFO  ruby 1.9.3 (2013-02-22) [x86_64-darwin12.2.1]
[2013-04-29 10:21:05] INFO  WEBrick::HTTPServer#start: pid=87936 port=3000
```

Acceptance Testing w/ Capybara
Testing a Rails Application

FOLLOW ALONG

- Let's examine the application by opening the URL http://localhost:3000 on a capable browser
- All pages are locked until we register on the Sign Up Page (provided by devise):



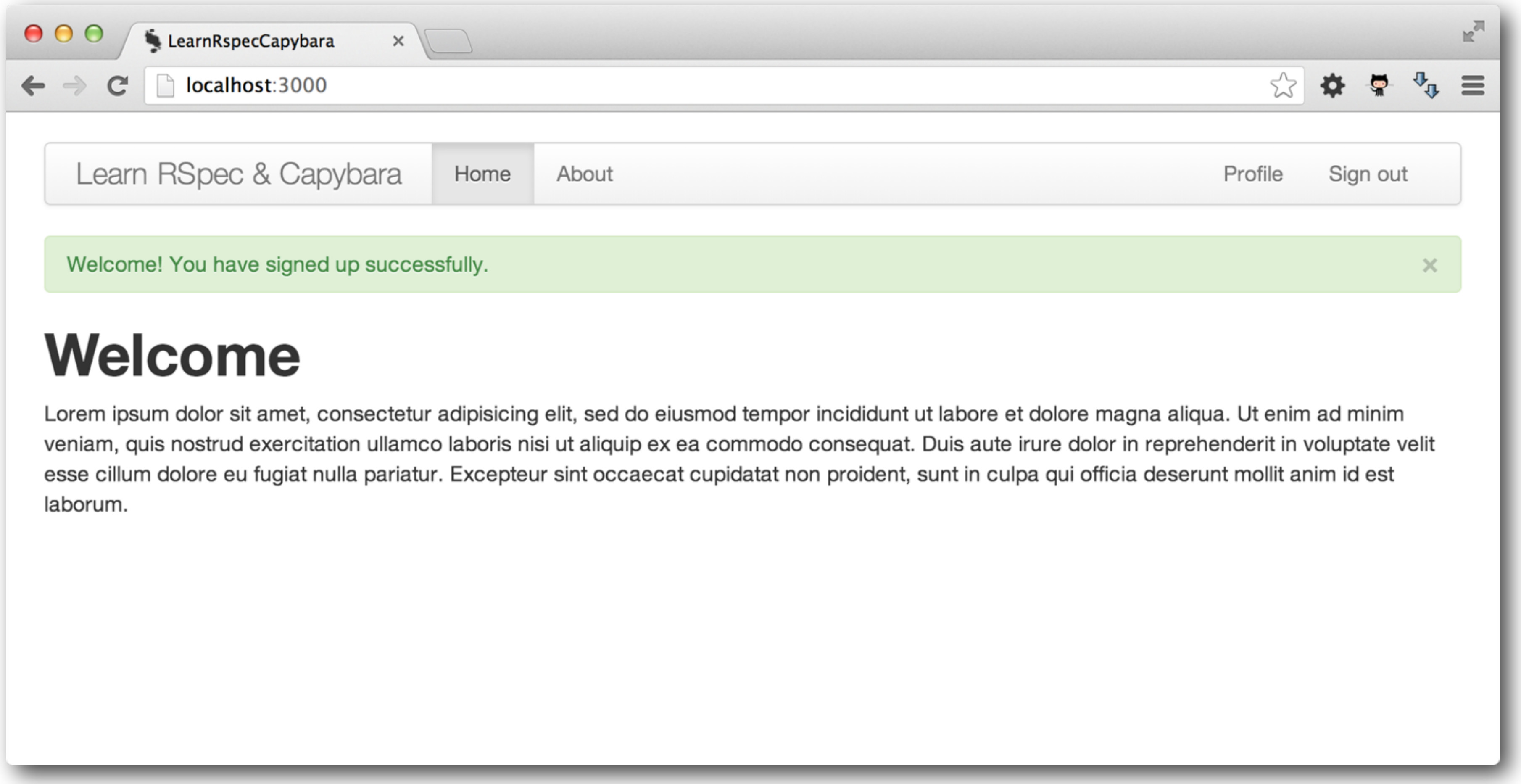
Sign Up Page - /users/sign_up

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- Once you are signed up you are redirected to the home page and shown a flash message:



Welcome! You have signed up successfully.

Welcome

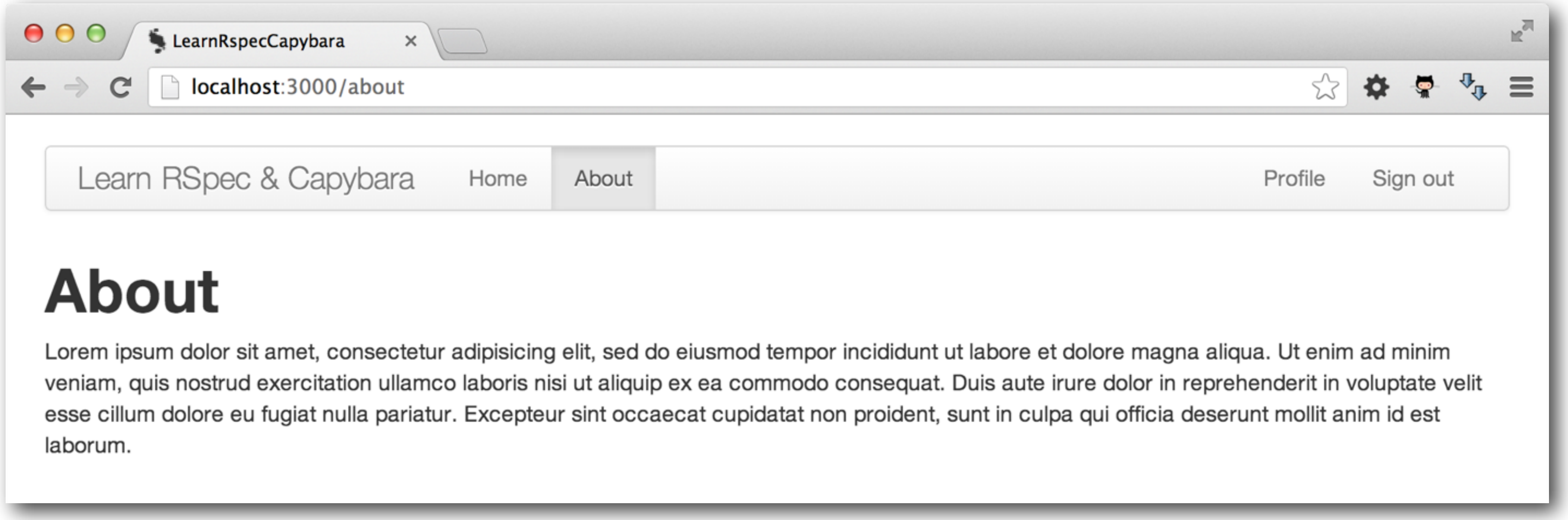
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Home Page - /

Acceptance Testing w/ Capybara
Testing a Rails Application

Follow Along

- About and home pages are ERB templates served by **high_voltage**:



About

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

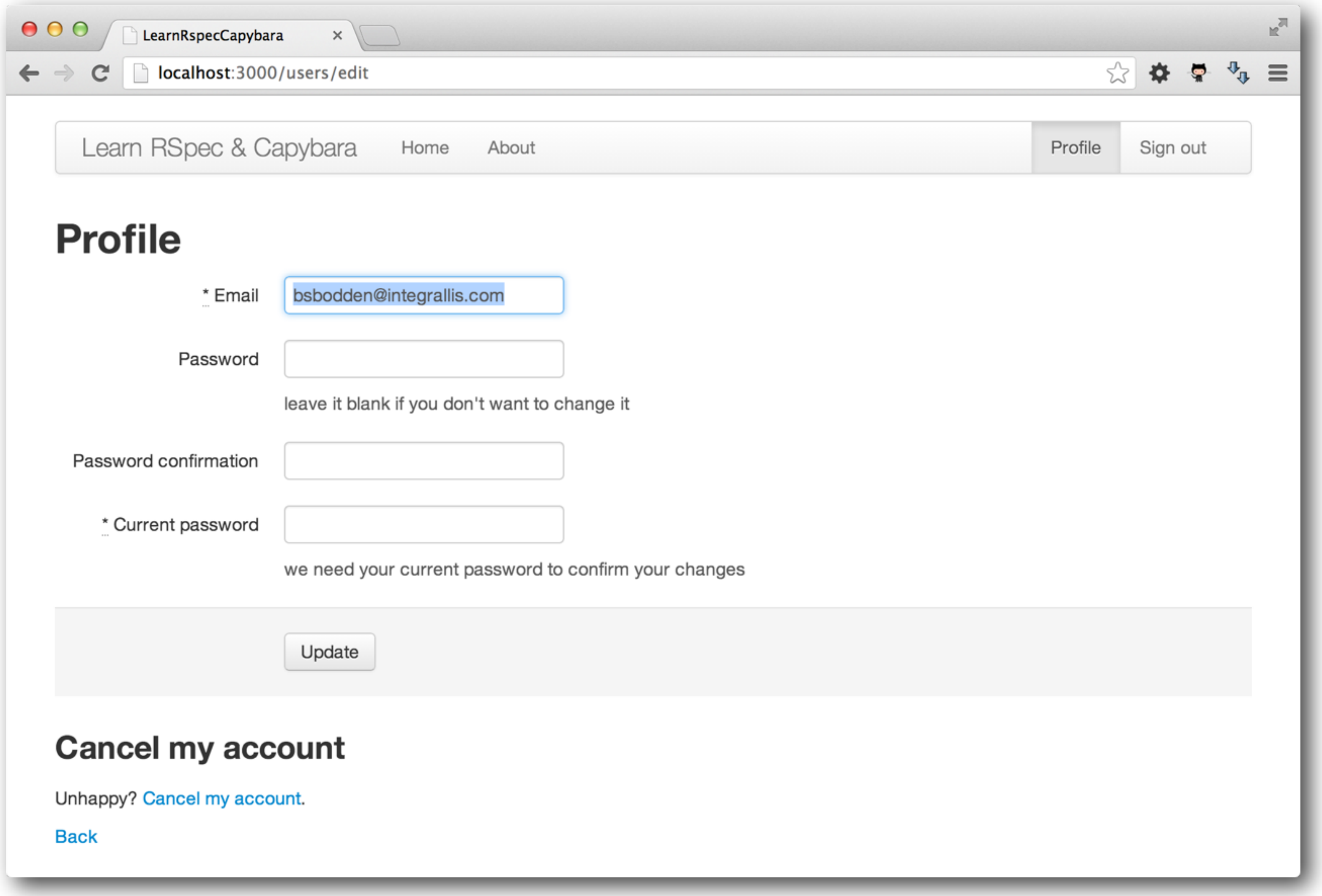
About Page - /about

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- The user profile page (provided by devise):



LearnRspecCapybara

localhost:3000/users/edit

Profile

* Email: bsbodden@integrallis.com

Password

leave it blank if you don't want to change it

Password confirmation

* Current password

we need your current password to confirm your changes

Cancel my account

Unhappy? Cancel my account.

Back

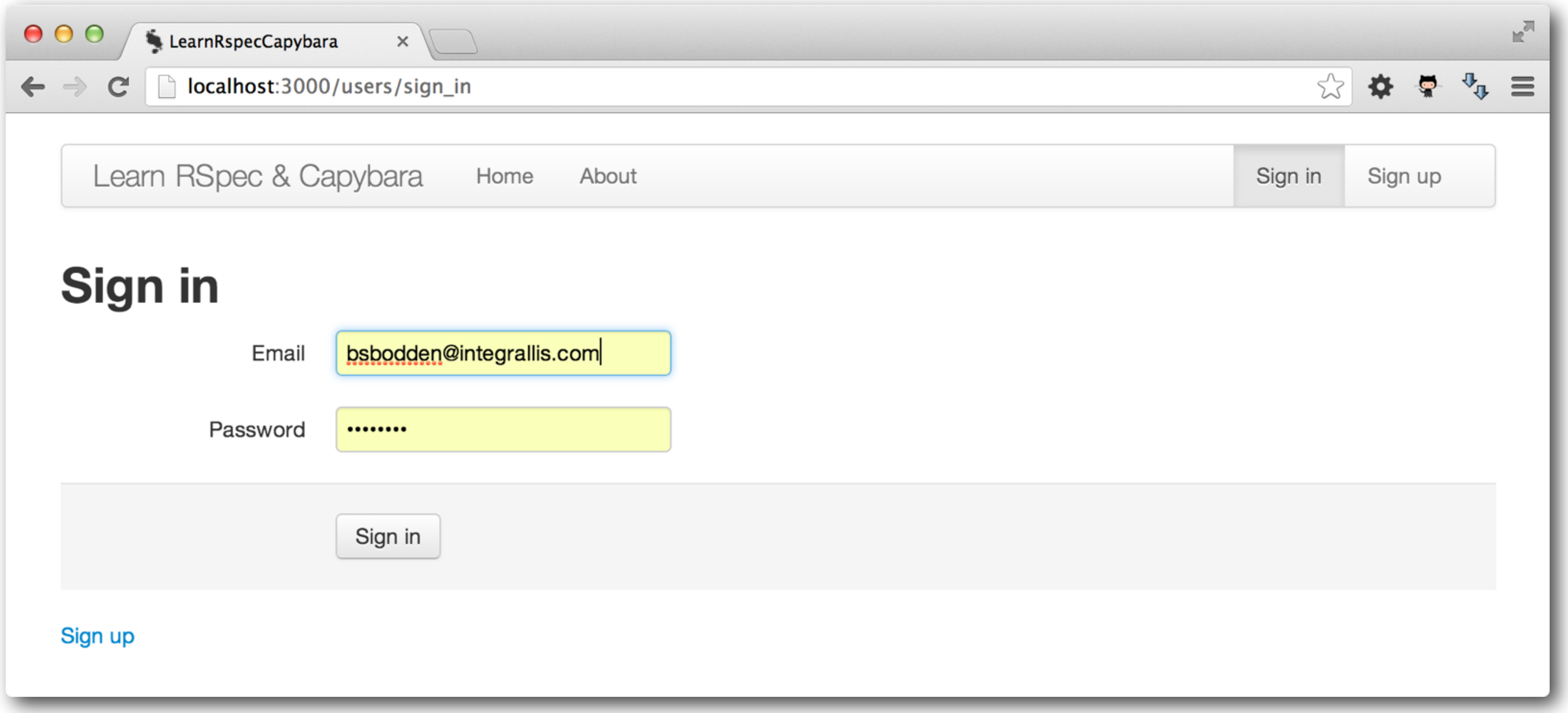
User Profile Page - /users/edit

Acceptance Testing w/ Capybara

Testing a Rails Application

FOLLOW ALONG

- The Sign In Page (also provided by devise):


LearnRspecCapybara

localhost:3000/users/sign_in

Sign in

Email: bsbodden@integrallis.com

Password: •••••••••

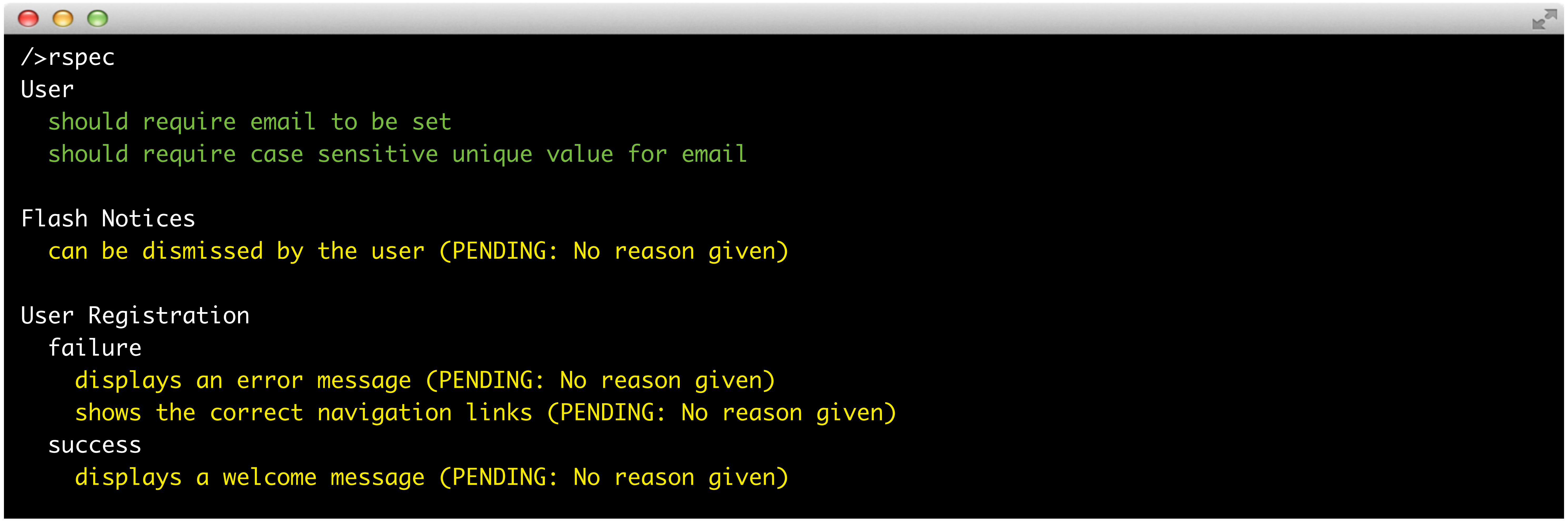
Sign In Page - /users/sign_in

Acceptance Testing w/ Capybara

Testing a Rails Application

FOLLOW ALONG

- If we run the specs using **rspec** you'll see **two passing specs** and the **rest as pending**:



```
/>rspec
User
  should require email to be set
  should require case sensitive unique value for email

Flash Notices
  can be dismissed by the user (PENDING: No reason given)

User Registration
  failure
    displays an error message (PENDING: No reason given)
    shows the correct navigation links (PENDING: No reason given)
  success
    displays a welcome message (PENDING: No reason given)
```

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- Let's examine the two passing specs:

```ruby
require 'spec_helper'

describe User do
  it { should validate_presence_of(:email) }
  it { should validate_uniqueness_of(:email) }
end
```

- They're just simple validation messages via **shoulda-matchers**

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- Let's start by tacking the Sessions Spec, specifically a successful login:

```ruby
context "success" do
  before do
    # sign in
  end

  it "displays a welcome message" do
    pending
  end

  it "shows the correct navigation links" do
    # should not see 'Sign in' and 'Sign up'
    # should see 'Profile' or 'Sign out'
    pending
  end
end
```

/spec/features/sessions_spec.rb

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- In the before block we'll fill in the email and passwords field and click the sign in button:

```
context "success" do
  before do
    fill_in 'Email', with: email
    fill_in 'Password', with: password
    click_button 'Sign in'
  end

...
end
```

Change and Run your specs!

/spec/features/sessions_spec.rb

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- Capybara provides us with the "page" object which we can inspect, for example, with the have_content method:

```
it "displays a welcome message" do
  expect(page).to have_content('Signed in successfully.')
end
```

Change and Run your specs!

/spec/features/sessions_spec.rb

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- We can also check for certain links to be or not be present in the page:

```ruby
it "shows the correct navigation links" do
  within('.navbar') do
    expect(page).to have_link('Profile')
    expect(page).to have_link('Sign out')
    expect(page).to_not have_link('Sign in')
    expect(page).to_not have_link('Sign up')
  end
end
```

Change and Run your specs!

/spec/features/sessions_spec.rb

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- Next let's tackle the flash notices spec. Notice that in the before block we use the visit method with a Rails named route:

```ruby
require 'spec_helper'

describe "Flash Notices", js: true do
  before do
    # When an unauthenticated user visit the edit_user_registration_path they
    # are redirected with a flash notice
    visit edit_user_registration_path
  end

  it "can be dismissed by the user" do
    # check that the flash message exists click to close the flash message
    # check that the flash message is gone
    pending
  end
end
```

/spec/features/flash_notices.rb

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- First, we'll check that the text of the flash message has appeared on the page:

```ruby
it "can be dismissed by the user" do
  expect(page).to have_content("You need to sign in or sign up before continuing.")
end
```

/spec/features/flash_notices.rb

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- Next will, find the alert div using the class CSS selector
- Inside of the alert div will find the close HREF and click it

```ruby
it "can be dismissed by the user" do
  expect(page).to have_content("You need to sign in or sign up before continuing.")

  within('.alert') do
    find('.close').click
  end
end
```

/spec/features/flash_notices.rb

Acceptance Testing w/ Capybara

Testing a Rails Application

Follow Along

- Now, we check that the content of the flash alert is no longer on the page:

```ruby
it "can be dismissed by the user" do
  expect(page).to have_content("You need to sign in or sign up before continuing.")

  within('.alert') do
    find('.close').click
  end

  expect(page).to_not have_content("You need to sign in or sign up before continuing.")
end
```

/spec/features/flash_notices.rb

Lab 2.0
Capybara

LAB 1.0

- In Lab 2.0, complete the remaining acceptance specs:
  - **The Registrations Spec** in /spec/features/registrations_spec.rb
  - **The Cancel Registration Spec** in /spec/features/cancel_registration.rb

Conclusions
Practice, Practice, Practice

- Don’t take TDD & BDD as dogma. Find ways to make it work for you!
- I don’t always use TDD & BDD but when I do …
- If you can TDD/BDD keep the code local until you can check it in with a corresponding test

Thanks

http://integrallis.com