videos

Ruby for Newbies: Testing with Rspec

Andrew Burgess on Aug 10th 2011 with 28 Comments

Tutorial Details

-
- **Topic**: Ruby, Rspec
- **Difficulty**: Easy
- **Estimated Completion Time**: 30 minutes

*This entry is part 13 of 13 in the **Ruby for Newbies** Session - Show All*
« Previous

Ruby is a one of the most popular languages used on the web. We're running a Session here on Nettuts+ that will introduce you to Ruby, as well as the great frameworks and tools that go along with Ruby development. In this episode, you'll learn about testing your Ruby code with Rspec, one of the best testing libraries in the business.

Prefer a Screencast?

Look Familiar?

If you've read my recent tutorial on JasmineJS, you'll probably notice several similarities in Rspec. Actually, the similarities are in Jasmine: Jasmine was created with Rspec in mind. We're going to look at how to can use Rspec to do TDD in Ruby. In this tutorial, we'll be creating some contrived Ruby classes to get us familiar with the Rspec syntax. However, the next "Ruby for Newbies" episode will feature using Rspec in conjunction withe some other libraries to test web apps … so stay tuned!

Setting Up

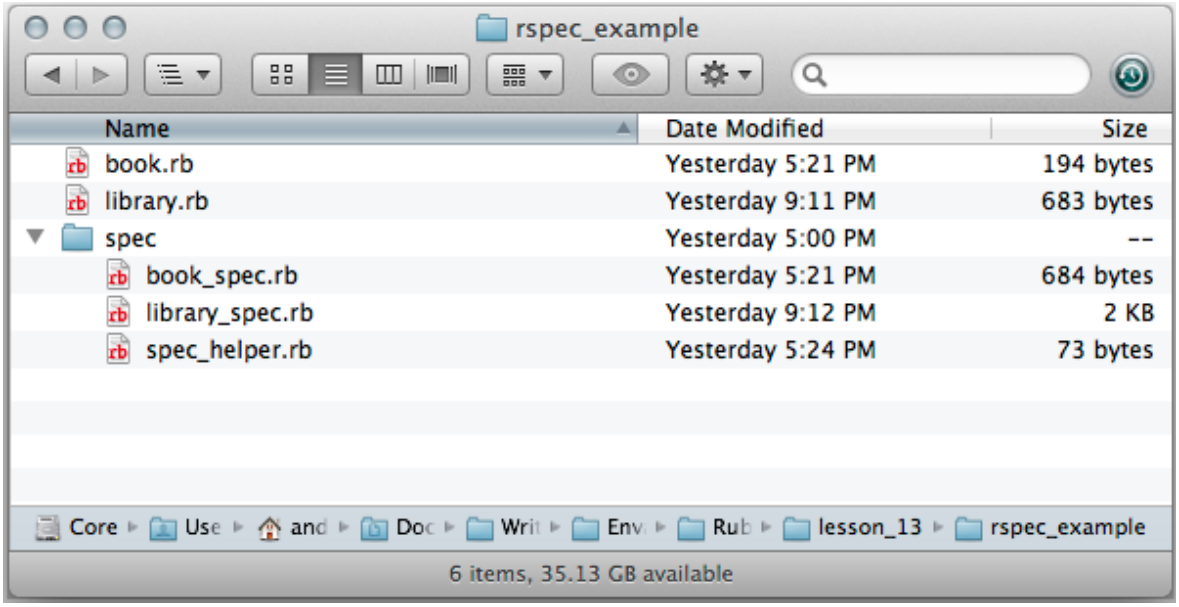
It's pretty easy to install Rspec. Pop open that command line and run this:

```
1 gem install rspec
```

That easy.

Now, let's set up a small project. We're going to create two classes: `Book` and `Library`. Our `Book` objects will just store a title, author, and category. Our `Library` object will store a list of books, save them to a file, and allow us to fetch them by category.

Here's what your project directory should look like:



We put the specifications (or specs) in a `spec` folder; we have one spec file for each class. Notice the `spec_helper.rb` file. For our specs to run, we need to `require` the Ruby classes we're testing. That's what we're doing inside the `spec_helper` file:

```
1  require_relative '../library'
2  require_relative '../book'
3
4  require 'yaml'
```

(Have you met `require_relative` yet? No? Well, `require_relative` is just like `require`, except that instead of searching your Ruby path, it searches *relative* to the current directory.)

You may not be familiar with the YAML module; YAML is a simple text database that we'll use to store data. You'll see how it works, and we'll talk more about it later.

So, now that we're all set up, let's get cracking on some specs!

The `Book` Class

Let's start with the tests for the `Book` class.

```
1  require 'spec_helper'
2
3  describe Book do
4
5  end
```

This is how we start: with a `describe` block. Our parameter to `describe` explains what we're testing: this could be a string, but in our case we're using the class name.

So what are we going to put inside this `describe` block?

```
1  before :each do
2      @book = Book.new "Title", "Author", :category
3  end
```

We'll begin by making a call to `before`; we pass the symbol `:each` to specifc that we want this code run before each test (we could also do `:all` to run it once before all tests). What exactly are we doing before each test?

We're creating an instance of `Book`. Notice how we're making it an instance variable, by prepending the variable name with `@`. We need to do this so that our variable will be accessible from within our tests. Otherwise, we'll just get a local variable that's only good inside the `before` block … which is no good at all.

Moving on,

```
describe "#new" do
    it "takes three parameters and returns a Book object" do
        @book.should be_an_instance_of Book
    end
end
```

Here's our first test. We're using a nested `describe` block here to say we're describing the actions of a specific method. You'll notice I've used the string "#new"; it's a convention in Ruby to talk refer to instance methods like this: `ClassName#methodName` Since we have the class name in our top-level `describe`, we're just putting the method name here.

Our test simply confims that we're indeed made a Book object.

Notice the grammar we use here: `object.should do_something`. Ninety-nine percent of your tests will take this form: you have an object, and you start by calling `should` or `should_not` on the object. Then, you pass to that object the call to another function. In this case that's `be_an_instance_of` (which takes `Book` as its single parameter). Altogether, this makes a perfectly readable test. It's very clear that `@book` should be an instance of the class `Book`. So, let's run it.

Open your terminal, `cd` into the project directory, and run `rspec spec`. The `spec` is the folder in which `rspec` will find the tests. You should see output saying something about "uninitialized constant Object::Book"; this just means there's no `Book` class. Let's fix that.

According to TDD, we only want to write enough code to fix this problem. In the `book.rb` file, that would be this:

```
class Book

end
```

Re-run the test (`rspec spec`), and you'll find it's passing fine. We don't have an `initialize` method, so calling `Ruby#new` has no effect right now. But, we can create `Book` objects (albeit hollow ones.) Normally, we would follow this process through the rest of our development: write a test (or a few related tests), watch it fail, make it pass, refactor, repeat. However, for this tutorial, I'll just show you the tests and code, and we'll discuss them.

So, more tests for `Book`:

```
describe "#title" do
    it "returns the correct title" do
        @book.title.should eql "Title"
    end
end
describe "#author" do
    it "returns the correct author" do
        @book.author.should eql "Author"
    end
end
describe "#category" do
    it "returns the correct category" do
        @book.category.should eql :category
    end
end
```

There should all be pretty strightforward to you. But notice how we're comparing in the test: with `eql`. There are three ways to test for equality with Rspec: using the operator == or the method `eql` both return `true` if the two objects have the same content. For example, both are strings or symbols that say the same thing. Then there's `equal`, which only returns true in the two objects are really and truely equal, meaning they are the same object in memory. In our case, `eql` (or ==) is what we want.

These will fail, so here's the code for `Book` to make them pass:

```
1  class Book
2      attr_accessor :title, :author, :category
3          def initialize title, author, category
4              @title = title
5              @author = author
6              @category = category
7          end
8  end
```

Let's move on to `Library`!

Speccing out the `Library` class

This one will be a bit more complicated. Let's start with this:

```
1  require 'spec_helper'
2
3  describe "Library object" do
4
5      before :all do
6          lib_obj = [
7              Book.new("JavaScript: The Good Parts", "Douglas Crockf
8              Book.new("Designing with Web Standards", "Jeffrey Zeld
9              Book.new("Don't Make me Think", "Steve Krug", :usabili
10             Book.new("JavaScript Patterns", "Stoyan Stefanov", :de
11             Book.new("Responsive Web Design", "Ethan Marcotte", :d
12         ]
13         File.open "books.yml", "w" do |f|
14             f.write YAML::dump lib_obj
15         end
16     end
17
18     before :each do
```

```
        @lib = Library.new "books.yml"
    end

end
```

This is all set-up: we're using two `before` blocks: one for `:each` and one for `:all`. In the `before :all` block, we create an array of books. Then we open the file "books.yml" (in "w"rite mode) and use `YAML` to dump the array into the file.

Short rabbit-trail to explain YAML a bit better: YAML is, according to the site "a human friendly data serialization standard for all programming languages." It's like a text-based database, kinda like JSON. We're importing YAML in our `spec_helper.rb`. The `YAML` module has two main methods you'll use: `dump`, which outputs the serialized data as a string. Then, `load` takes the data string and coverts it back to Ruby objects.

So, we've created this file with some data. Before `:each` test, we're going to create a `Library` object, passing it the name of the YAML file. Now let's see the tests:

```
describe "#new" do

    context "with no parameters" do
        it "has no books" do
            lib = Library.new
            lib.should have(0).books
        end
    end
    context "with a yaml file parameter" do
        it "has five books" do
            @lib.should have(5).books
        end
    end
end

it "returns all the books in a given category" do
    @lib.get_books_in_category(:development).length.should == 2
end

it "accepts new books" do
    @lib.add_book( Book.new("Designing for the Web", "Mark Boulton
    @lib.get_book("Designing for the Web").should be_an_instance_c
end

it "saves the library" do
    books = @lib.books.map { |book| book.title }
    @lib.save
    lib2 = Library.new "books.yml"
    books2 = lib2.books.map { |book| book.title }
    books.should eql books2
end
```

We start with an inner `describe` block especially for the `Library#new` method. We're introducing another block here: `context` This allows us to specify a context for tests inside it, or spec out different outcomes for diffenent situations. In our example, we have two different context: "with no parameters" and "with a yaml file parameter"; these show the two behaviours for using `Library#new`.

Also, notice the test matchers we're using in these two tests: `lib.should have(0).books` and `@lib.should have(5).books`. The other way to write this would be `lib.books.length.should == 5`, but this isn't as readable. However, it shows that we need to have a `books` property that is an array of the books we have.

Then, we have three other tests to test the functionality of getting books by category, adding a book to the library, and saving the library. These are all failing, so let's write the class now.

```
class Library
    attr_accessor :books

    def initialize lib_file = false
        @lib_file = lib_file
        @books = @lib_file ? YAML::load(File.read(@lib_file)) : []
    end

    def get_books_in_category category
        @books.select do |book|
            book.category == category
        end
    end

    def add_book book
        @books.push book
    end

    def get_book title
        @books.select do |book|
            book.title == title
        end.first
    end

    def save lib_file = false
        @lib_file = lib_file || @lib_file || "library.yml"
        File.open @lib_file, "w" do |f|
            f.write YAML::dump @books
        end
    end
end
```

We could write up more tests and add a lot of other functionality to this `Library` class, but we'll stop there. Now running `rspec spec`, you'll see that all the tests pass.

```
→ rspec spec
.........

Finished in 0.01005 seconds
9 examples, 0 failures
```

This doesn't give us that much information about the tests, though. If you want to see more, use the nested format parameter: `rspec spec --format nested`. You'll see this:

```
→ rspec spec --format nested

Book
  #new
    takes three parameters and returns a Book object
  #title
    returns the correct title
  #author
    returns the correct author
  #category
    returns the correct category

Library object
  returns all the books in a given category
  accepts new books
  saves the library
  #new
    with no parameters
      has no books
    with a yaml file parameter
      has five books

Finished in 0.01064 seconds
9 examples, 0 failures
```

A Few Last Matchers

Before we wrap up, let me show you a couple of other matchers

- `obj.should be_true`, `obj.should be_false`, `obj.should be_nil`, `obj.should be_empty` – the first three of these could be done by `== true`, etc. `be_empty` will be true if `obj.empty?` is true.
- `obj.should exist` – does this object even exist yet?
- `obj.should have_at_most(n).items`, `object.should have_at_least(n).items` – like `have`, but will pass if there are more or fewer than `n` items, respectively.
- `obj.should include(a[,b,...])` – are one or more items in an array?
- `obj.should match(string_or_regex)` – does the object match the string or regex?
- `obj.should raise_exception(error)` – does this method raise an error when called?
- `obj.should respond_to(method_name)` – does this object have this method? Can take more than one method name, in either strings or symbols.

Want to Learn More?

Rspec is one of the best frameworks for testing in Ruby, and there's a ton you can do with it. To learn more, check out the Rspec website. There's also the The Rspec book, which teaches more than just Rspec: it's all about TDD and BDD in Ruby. I'm reading it now, and it is extremely thorough and in-depth.

Well, that's all for this lesson! Next time, we'll look at how we can use Rspec to test the interfaces in a web app.

Tags:RubyVideos

By Andrew Burgess

Hi! I'm Andrew Burgess, a Staff Writer here on Nettuts+. I've been hanging around the Nettuts+ since early 2009; I discovered the site when I was looking for an introduction to jQuery. Since discovering the site, my web development skills have skyrocketed; I think that's the default experience! Now, I've been writing for Nettuts+ regularly since late 2009.I've been working with the computers since I was pretty young, and with the web since 2006. I've dabbled with over a dozen programming languages, but I'm most comfortable in JavaScript and Ruby. Currently, I'm a university student, studying computer science.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more