**Test Driven Development – Rspec**

RSpec is a testing tool for Ruby, created for BDD (behaviour-driven development). It’s the most frequently used testing library for Ruby in application production. Though it has a very rich and powerful DSL (domain-specific language), its core is a simple tool which is rather easy to use.

To understand the reason RSpec is the way it is, you need to understand the whole point of BDD and its parent TDD (Test-Driven Development). Kent Beck in his 2000 book Extreme Programming Explained first bought TDD to the wider audiences. Instead of writing tests for some code that you already have, you work in a red-green loop.

1. Write the smallest possible test case that matches what we need to program.
2. Run the test and watch it fail, this triggers you into thinking how to write only the code that makes it pass.
3. Write some code with the goal of making the test pass.
4. Run your test suite. Repeat steps 3 and 4 till it passes all test.
5. Go back and refactor your code, making it as simple and clear as possible while keeping the test green.

A problem commonly faced by newcomers when starting to test is falling into the trap of writing tests that do too much, test too little and require deep concentration in order to know what’s going on.

For example:

def test\_making\_order

dvd = dvd.new(:title => "Test Driven Development - RSpec", :price => 35)

customer = Customer.new

order = Order.new(customer, dvd)

order.submit

assert(customer.orders.last == order)

assert(customer.ordered\_dvds.last == book)

assert(order.complete?)

assert(!order.shipped?)

end

With RSpec we can get a little over the top, for example:

describe Order do

describe "#submit" do

before do

@dvd = dvd.new(:title => "Test Driven Development - RSpec", :price => 25)

@customer = Customer.new

@order = Order.new(@customer, @dvd)

@order.submit

end

describe "customer" do

it "puts the ordered dvd in customer's order history" do

expect(@customer.orders).to include(@order)

expect(@customer.ordered\_dvds).to include(@dvd)

end

end

describe "order" do

it "is marked as complete" do

expect(@order).to be\_complete

end

it "is not yet shipped" do

expect(@order).not\_to be\_shipped

end

end

end

end

### **Setting Up RSpec**

Let's start a new Ruby project where we'll configure RSpec as a dependency via Bundler. Create a new directory and put the following code in your Gemfile:

*# Gemfile*

source "https://rubygems.org"

gem "rspec"

Open your project's directory in your terminal, and type bundle install --path .bundle to install the latest version of RSpec and all related dependencies. You'll see output similar to the one below:

$ bundle install --path .bundle

Fetching gem metadata from https://rubygems.org/.........

Resolving dependencies...

Installing diff-lcs 1.2.5

Installing rspec-support 3.1.2

Installing rspec-core 3.1.7

Installing rspec-expectations 3.1.2

Installing rspec-mocks 3.1.3

Installing rspec 3.1.0

Using bundler 1.6.0

Your bundle is complete!

It was installed into ./.bundle

### **Writing the First Spec**

By convention, tests written with RSpec are called "specs" (short for "specifications") and are stored in the project's spec directory. Create that directory in your project too:

mkdir spec

Let's begin writing our first spec.

*# spec/string\_calculator\_spec.rb*

describe StringCalculator **do**

**end**

With RSpec, we are always describing the behavior of classes, modules and their methods. The describe block is always used at the top to put specs in a context. It can accept either a class name, in which case the class needs to exist, or any string you'd like.

Since Ruby methods do not require the use of parenthesis, this file already begins to feel more like an essay, rather than computer code. That's exactly the goal.

To run the specs, type:

bundle exec rspec

Do this now, and our spec will fail with the uninitialized constant StringCalculator (NameError) error. That's expected, as we haven't created that class yet.

Create a new directory called lib:

mkdir lib

Declare StringCalculator in string\_calculator.rb:

*# lib/string\_calculator.rb*

**class** **StringCalculator**

**end**

And require it in your spec:

*# spec/string\_calculator\_spec.rb*

require "string\_calculator"

describe StringCalculator **do**

**end**

Running RSpec now passes:

$ bundle exec rspec

No examples found.

Finished in 0.00068 seconds (files took 0.30099 seconds to load)

0 examples, 0 failures

That means we are ready to add code.

The simplest thing our string calculator can do is accept an empty string, in which case we might decide we want it to return a zero. The method we need to describe first is add.

*# spec/string\_calculator\_spec.rb*

describe StringCalculator **do**

describe ".add" **do**

context "given an empty string" **do**

it "returns zero" **do**

expect(StringCalculator.add("")).to eql(0)

**end**

**end**

**end**

**end**

There are a couple of new things here:

* We are using another describe block to describe the add class method. By convention, class methods are prefixed with a dot (".add"), and instance methods with a dash ("#add").
* We are using a context block to describe the context under which the add method is expected to return zero. context is technically the same as describe, but is used in different places, to aid reading of the code.
* We are using an it block to describe a specific example, which is RSpec's way to say "test case". Generally, every example should be descriptive, and together with the context should form an understandable sentence. This one reads as "add class method: given an empty string, it returns zero".
* expect(...).to and the negative variant expect(...).not\_to are used to define expected outcomes. The Ruby expression they are given (in our case,StringCalculator.add("")) is combined with a matcher to fully define an expectation on a piece of code. The matcher we are using here is eql, a basic equality matcher. RSpec comes with many more matchers.

If we run our spec now, we will get a failure that the method is not defined:

$ bundle exec rspec

F

Failures:

1) StringCalculator.add given an empty string returns zero

Failure/Error: expect(StringCalculator.add("")).to eql(0)

NoMethodError:

undefined method `add' for StringCalculator:Class

# ./spec/string\_calculator\_spec.rb:8:in `block (4 levels) in <top (required)>'

Let's make that spec pass:

*# lib/string\_calculator.rb*

**class** **StringCalculator**

**def** **self**.add(input)

0

**end**

**end**

We want to write the simplest possible code which will make the specs pass, remember? If you run bundle exec rspec now, the spec does pass.

### **Towards Working Code**

Our next assignment is to make the calculator work given a single number in a string. Let's write some examples for that:

*# spec/string\_calculator\_spec.rb*

describe StringCalculator **do**

describe ".add" **do**

context "given '4'" **do**

it "returns 4" **do**

expect(StringCalculator.add("4")).to eql(4)

**end**

**end**

context "given '10'" **do**

it "returns 10" **do**

expect(StringCalculator.add("10")).to eql(10)

**end**

**end**

**end**

**end**

After we have run the specs, we will get some helpful output:

$ bundle exec rspec

.FF

Failures:

1) StringCalculator.add given '4' returns 4

Failure/Error: expect(StringCalculator.add("4")).to eql(4)

expected: 4

got: 0

(compared using eql?)

# ./spec/string\_calculator\_spec.rb:14:in `block (4 levels) in <top (required)>'

2) StringCalculator.add given '10' returns 10

Failure/Error: expect(StringCalculator.add("10")).to eql(10)

expected: 10

got: 0

(compared using eql?)

# ./spec/string\_calculator\_spec.rb:20:in `block (4 levels) in <top (required)>'

Finished in 0.00133 seconds (files took 0.0835 seconds to load)

3 examples, 2 failures

Again, our goal is to make them pass:

*# lib/string\_calculator.rb*

**class** **StringCalculator**

**def** **self**.add(input)

**if** input.empty?

0

**else**

input.to\_i

**end**

**end**

**end**

Time to make the calculator actually do some math. Let's write some examples based on strings containing comma-separated numbers. It might make sense to introduce a nested context, "two numbers":

*# spec/string\_calculator\_spec.rb*

describe StringCalculator **do**

describe ".add" **do**

context "two numbers" **do**

context "given '2,4'" **do**

it "returns 6" **do**

expect(StringCalculator.add("2,4")).to eql(6)

**end**

**end**

context "given '17,100'" **do**

it "returns 117" **do**

expect(StringCalculator.add("17,100")).to eql(117)

**end**

**end**

**end**

**end**

**end**

These specs fail, as you'd expect. The full output is omitted here for brevity, but you are encouraged to run bundle exec RSpec after every change in code. Here's one way to make the specs pass:

**class** **StringCalculator**

**def** **self**.add(input)

**if** input.empty?

0

**else**

numbers = input.split(",").map { |num| num.to\_i }

numbers.inject(0) { |sum, number| sum + number }

**end**

**end**

**end**

RSpec has more than one way to display its output. A very popular alternative to the default dot format is the "documentation" format:

$ bundle exec rspec --format documentation

StringCalculator

.add

given an empty string

returns zero

single numbers

given '4'

returns 4

given '10'

returns 10

two numbers

given '2,4'

returns 6

given '17,100'

returns 117

We’ve just covered the basics of working with RSpec. When you browse RSpec's built-in matchers, you will be ready to start writing your first tests.