RSpec

Walter Schlosser

March 17, 2015

Intro to RSpec

RSpec is a Ruby tool for *Behavior Driven Development*. Behavior driven development is a way to think about and use test driven development at a higher level. One of the issues with test driven development is that tests usually call a function and then assert that the returned value is correct. This encourages low level testing of individual functions, and can cause the programmer to forget about testing the high level, overall functionality the code should have. Behavior driven development is about writing tests for software functionality, and not just functions. RSpec allows BDD in a very readable manner, and allows software testing to resemble a conversation, such as "Describe how a bowling game scores a match where all balls are gutters. It should have a score of zero."

BDD or TDD?

Programmers differ in opinion over which methodology is better. Some argue that BDD produces fewer bugs/errors than TDD since the overall picture is kept in mind, while others say BDD is just TDD with a wrapper. In the end, both software development methodologies are still about designing before writing too much code, and having a test suite to alert the programmer of any code changes that just broke the software.

Installing and using RSpec

RSpec is a gem, and therefore can be installed system-wide with *sudo gem install rspec*, or declared in a gemfile for bundler. After installed, RSpec runs at the command line such as *rspec mySpecFile.rb*. RSpec is an internal DSL, and therefore directly executes the code in the file passed. For this reason, the file can be named anything as long as the code inside is valid for RSpec. For example, *rspec class.spec* is valid. There are many options, but one useful one is *rspec file.spec -fdoc*, which prints the list of tests in a readable way before printing the results like usual.

Running the demo

The included demo has files *classDefs.rb* and *classSpecs.rb*. Code implementation to be tested is in *classDefs.rb*, while the RSpec code is in *classSpecs.rb*. To run the test suite, use *rspec classSpecs.rb -fdoc* where the *-fdoc* is optional but recommended since it prints a readable output to show how RSpec interpreted the tests. The class and test implementations are made to show one pending test example and one failing test example while the rest pass.

Structure of an RSpec test suite

See lines 10 - 86 in classSpecs.rb
RSpec attempts to describe behavior in the most readable way possible by using groups, contexts, and examples. A group is a set of tests that describe how the passed class should behave. The describe function, used as *RSpec.describe MyClass do ... end*, opens a group for the passed class. A block is passed to the group where contexts will be defined. A context is a description of the situation the class is in. For example, an Array can be empty, or have many elements. The context function is used as *context "description" do ... end* where the string is used to describe the context in English, and examples (tests) will be defined in the block. An example describes the result to be expected, and then executes the code in the passed block. Examples are called as *it "should do this" do ... end* where we write setup code and then a call to *expect* inside the block. A call to *expect* takes an expression as an argument, allows the *to* function to be called on it, and then takes a condition. Some conditions are *eq(...)* for equal, and *be_truthy/be_falsey* for bool expressions. For example, *expect(1+1).to eq(2)* and *expect(1 == 1).to be_truthy* are valid. Refer to the demo code lines stated at the beginning of this section to see the full test group structure. This nested structure is what allows the *-fdoc* option at the command line to display the tests in a readable manner. RSpec allows a test to pass, fail, or remain pending. Pass and fail are self-explanatory. A pending test is expected to fail by the programmer for any reason they define. The function *pending "reason"* is called inside an example if it should be marked as such. Any examples without an associated block to run are also implicitly marked as pending.

Hooks

See lines 90 - 131 in classSpecs.rb
When testing, some code such as instantiating a new object must be done before every test. RSpec provides the *before* and *after* helper functions to repeat code. Both function take a symbol of either :context, :example, or :suite to define which portion of the testing structure they should run before. The simple way to use them is to write a line such as *before(:context) do ... end* right inside the group opening. However, sometimes the before code

should run before every context that requires it, and not others. This is accomplished with a filter. A filter is passed as a hash to the *before/after* call, like *before(:context), :type⇒:model do ... end.* This code will run before any context call that also passes this hash, *context "description", :type⇒:model do ... end*, and not the others. To use a filter with *before/after*, they must be defined in a configuration block for RSpec before creating test groups. A configuration block is opened as *RSpec.configure do ... end.* Inside the block, *before/after* can be defined with or without filters. See lines 95 - 108 specifically in *classSpecs.rb* to see the configuration call.

Shared examples

See lines 136 - 170 in classSpecs.rb
RSpec encourages DRY code like much of the Ruby community. Sometimes two classes behave the same in many aspects, such as containers (Array, Set) or comparable objects. Instead of writing the same tests twice, they can be written once and included in any group they are needed in. These shared examples are defined in a block passed to the function *RSpec.shared_examples "a container, comparable, etc" do ... end.* These examples can then be included in a test group by using the helper function *it_behaves_like "a container, comparable, etc"* where the string passed much match a shared example package description. Shared examples are very useful for any classes that include the same module and therefore share some functionality.

RSpec with Rails

Note: The included demo does not use Rails so the following setup is not needed.
To use RSpec instead of the default test framework Rails uses, it must be defined in the gemfile for bundler to install. Add the code *group :development, :test do; gem 'rspec-rails'; end* and run bundle install. To setup RSpec files, run the command *rails generate rspec:install*, and then create spec files in the spec directory. Lastly, *bundle exec rspec* will run all tests in the spec directory that end in "_spec.rb". The setup tells Rails that RSpec is to be used, and so when generating models/controllers or even scaffolds, spec files will be generated instead of the default test files.

How I spent my time

0.5 hr - Getting RSpec up and running with the example given on the website, rspec.info
0.5 hr - Reading up on the similarities/differences between BDD and TDD
1 hr - Reading the docs and following along with the examples for basic structure of RSpec, relishapp.com/rspec/rspec-core/v/3-2/docs
1 hr - Reading about other helpful features of RSpec, like helpers and shared examples
0.5 hr - Reading how to use with Rails and trying it out
1 hr - Creating a demo file that uses most of what I learned about RSpec
1.5 hr - Writing up this report about the demo