RSPec Documentation

Contents:

Written by ,

Narayanan K

Gsoc-2010,OAR Testsuites

Introduction to Rspec

Rspec is a behavior driven test development framework developed for Ruby.

Behavior Driven Development(BDD) began its journey as an attempt to better understand and explain the process of Test Driven Development. Behavior driven development is an agile software development technique .

Lot of beautiful documentations describing RSpec are available online including the e-book, “The RSpec Book” by David Chelinsky.

I have included the sites and the e-book that is available in the reference section of this documentation that really got me testing my codes with RSpec.

This documentation specifically deals with how to perform unit/scenario testing of the OAR REST APIs.

The RSpec tool is a Ruby package that lets you build a specification alongside your software. This specification is actually a test that describes the behavior of your system.

Here's the flow for development with RSpec:

- You write a test. This test describes the behavior of a small element of your system.
- You run the test. The test fails because you have not yet built the code for that part of your system. This important step tests your test case, verifying that your test case fails when it should.
- You write enough code to make the test pass.
- You run the tests and verify that they pass.

In essence, an RSpec developer turns test cases from red (failing) to green (passing) all day. It's a motivating process. In this article, I walk you through working with the basics in RSpec.

Getting started:

To get started, I'll assume you've installed Ruby and gems. You'll also need to install RSpec.

Type:

```
gem install rspec
```

Terminology used in Rspec:

Subject code : The code whose behaviour we are specifying with Rspec.

Expectation : An expression of how the subject code is expected to behave

Code example : An executable example of how the subject code can be used, and its expected behaviour (expressed with expectations) in a given context. In BDD, we write the code examples before the subject code they document.

Example group : A group of code examples.

Spec, a.k.a. spec file : A file that contains one or more example groups.

If you already have some experience with Test::Unit or similar tools in other languages and/or TDD, the words we're using here map directly to words you're already familiar with:

- **Assertion** becomes **Expectation.**
- **Test Method** becomes **Code Example**.
- ***Test Case*** becomes ***Example Group***.

Writing Simple Rspec Tests :

A simple spec code to test if 1+2 is 3. The testfile name is : **simple_math_spec.rb**

```
require 'rubygems'
require 'spec'

describe "simple math" do

  it "should provide a sum of two numbers" do

  (1 + 2).should == 3
end
end
```

The 'should' statement verifies the assertion if it is successful or failure.

A general RSpec Test Structure with Before and After statements:

```
describe Thing do

  before(:all) do
    # This is run once and only once, before all of the examples
    # and before any before(:each) blocks.
  end

  before(:each) do
    # This is run before each example.
  end

  before do
    # :each is the default, so this is the same as before(:each)
  end

  it "should do stuff" do
    ...
  end

  it "should do more stuff" do
    ...
  end

  after(:each) do
    # this is after each example
  end

  after do
    # :each is the default, so this is the same as after(:each)
  end
```

```
  after(:all) do
    # this is run once and only once after all of the examples
    # and after any after(:each) blocks
  end

end
```

Warning: The use of `before(:all)` and `after(:all)` is generally discouraged because it introduces dependencies between the Examples. Still, it might prove useful for very expensive operations if you know what you are doing.

Note: before and after methods are similar to setup and teardown functionalities in Test::Unit

Helper Methods:

```
describe "..." do

  it "..." do
    helper_method
  end

  def helper_method
    ...
  end

end
```

Pending Examples:

Leave out the block:

```
it "should say foo"
```

The output will say PENDING (Not Yet Implemented).

Note : A well written article on how to write Rspec testcases for simulating a state machine and then write corresponding development code to pass each testcase can be found in the link :

http://www.ibm.com/developerworks/web/library/wa-rspec/

Running an Rspec Test :

Running our simple_math_spec.rb with the spec command:

$ spec simple_math_spec.rb

You should see output like this:

.

Finished in 0.00621 seconds
1 example, 0 failures

This is RSpec's default output format, the progress bar format. It prints out a dot for every code example that is executed and passes (only one in this case). If an example fails, it prints an F. If an example is pending it prints a *. These dots, F's and *'s are printed after each example is run, so when you have many examples you can actually see the progress of the run, hence the name "progress bar." After the progress bar, it prints out the time it took to run and then a summary of what was run. In this case, we ran one example and it passed, so there are no failures.

Now try running it with the ruby command instead:

$ ruby simple_math_spec.rb

You should see the same output. When executing individual spec files, the spec and ruby commands are somewhat interchangeable. We do, however, get some added value from the spec command when running more than just one file.

Other spec options:

$ spec test.rb --format specdoc *(To see test results in detail formatted form)*

$ spec path/to/my/specs --format html:path/to/my/report.html *(Redirects output to an html page)*

$ spec specdir --backtrace

$ spec specdir --color

Invoke With Options Stored in a File with --options

You can store any combination of these options in a file and tell the spec command where to find it. For example, you can add this to spec/spec.opts:

--color
--format specdoc

You can list as many options as you want, with one or more words per line. As long as there is a space, tab or newline between each word, they will all be parsed and loaded. Then you can run the code examples with this command:

$ spec specdir --options specdir/spec.opts

That will invoke the options listed in the file.

Scenario Testing Examples of OAR APIs using Rspec:

Rspec can be used to perform scenario testing.

Assume we have a library, librestapi.rb, that contains call to all the OAR REST APIs with error management.

Sample librestapi.rb with calls to get /jobs/<id>, post /jobs and delete /jobs

```
#############################################################################

# Method: specific_job_details(jobid)

# Input: jobid

# Result: GETs details of specific job & stores in hash specificjobdetails

#############################################################################

def specific_job_details(jobid)
@specificjobdetails = get(@api, "/jobs/#{jobid}")
if !@specificjobdetails.is_a?(Hash) or @specificjobdetails.empty?
        raise 'Error: In return value of GET /jobs/<jobid> API'
end
end

#############################################################################

# Method: submit_job(jhash)

# Input: jhash containing details of resources,jobscript in hash form

# Result: Returns the submitted job Details in Hash and stores in jobstatus

#############################################################################

def submit_job(jhash)
@jobstatus = post(@api, '/jobs', jhash)
if !@jobstatus.is_a?(Hash) or @jobstatus.empty?
        raise 'Error: In return value of POST /jobs API'
end
end
```

```
#############################################################################

# Method: del_job(jobid)

# Input:  jobid

# Result: Returns the deleted job Details in Hash and stores in deletestatus

#############################################################################

def del_job(jobid)
@deletestatus = post(@api,"/jobs/#{jobid}/deletions/new", '')
if !@deletestatus.is_a?(Hash) or @deletestatus.empty?
        raise 'Error: In return value of POST /jobs/<id>/deletions/new API'
end
end
```

Now we add librestapi.rb path to RUBYLIB variable so that we can import the lib into our spec files using 'require'

Here 2 example scenario testing of OAR APIs using Rspec are shown.

a. Submit a job, Check if the job is still running after 10 minutes. (testspec.rb)

```
require 'librestapi'
$jobid = ""

describe OarApis do

 before :all do

  # Custom variables
APIURI="http://www.grenoble.grid5000.fr/oarapi"

#Object of OarApis class
@obj = OarApis.new(APIURI)

end

#Scenario : Submit a Job, check if the job is running after 1 minute

#Test for Submitting a job

it "should submit a job successfully " do
resource = "/nodes=1/core=1"
```

```
script = "/home/nk/test.sh"     #This test must be running for more than or = 10minutes
walltime = "1"

jhash = { 'resource' => "#{resource}" , 'script' => "#{script}", 'walltime' =>
"#{walltime}" }

begin
        @obj.submit_job(jhash)
rescue
        puts "#{$!}"
        exit 2
end

$jobid = @obj.jobstatus['id'].to_s

 @obj.jobstatus['status'].to_s.should == "submitted"
end



 #Test if job is running after 1 minute

 it "should check if the submitted job is running after 60 seconds" do
sleep 60
begin
          @obj.specific_job_details($jobid)
rescue
          puts "#{$!}"
          exit 2
end

 @obj.specificjobdetails['state'].to_s.should == "Running"

end
end
```

Running using spec : ***$ spec testspec.rb --format specdoc***

```
OarApis
- should submit a job successfully
- should contain jobid in queue of created job
- should delete the currently submitted job using the post api and jobid
- should not contain the deleted job in the queue now

Finished in 38.062151 seconds
4 examples, 0 failures
```

b. Submit a Job, check the queue for that job, Delete the job and again test the queue list

The test elements are are as follows:

```
  #Submitting a job

  it "should submit a job successfully " do
@obj.submit_job

  $jobid = @obj.jobs['id'].to_s
@obj.jobs['status'].to_s.should == "submitted"
end

  #Checking the queue (Can use GET /jobs to check) immediately.

  it "should contain jobid in queue of created job" do
@obj.full_job_details
@obj.jobhash.each do |jhash|
if  jhash['job_id'] == $jobid
@c=1
end
end
@c.should == 1
end

  #Delete the job

 it "should delete the currently submitted job using the post api and jobid" do
@obj.del_job($jobid)
@obj.deletestatus['status'].should == "Delete request registered"
end

 #Check the queue to ensure the job deleted is no more there #Negative Test.
#But sleeps for sometime until the jobid is removed from the queue.

  it "should not contain the deleted job in the queue now" do
@c=0
sleep 45  # This is arbitrary as queue will be freed only based on its queue contents..
@obj.full_job_details
@obj.jobhash.each do |jhash|
if jhash['job_id'] == $jobid
@c=1
end
end
```

```
@c.should_not == 1
end
```

Creating Rspec Testsuites:

Running Several Specs at Once :

Running specs directly is handy if you just want to run one single file, but in most cases you really want to run many of them in one go. The simplest way to do this is to just pass the directory containing your spec files to the spec command. So if your spec files are in the specdir directory, you can just do this:

$ spec spec dir

In either case, the spec command will load all of the spec files in the spec directory and its sub-directories. By default, the spec command only loads files ending with _spec.rb.

Several other tools like Rake, Autotest, Heckle, Rcov, Testmate etc..are used along with Rspec for more specific result set for specific requirements.

Rspec has an Extension called “Cucumber” which is mostly built for higher level scenario testing. Visit http://cukes.info

References:

http://www.ibm.com/developerworks/web/library/wa-rspec/

http://media.pragprog.com/titles/achbd/examples.pdf

http://rspec.info/documentation/

http://www.oreillynet.com/pub/a/ruby/2007/08/09/behavior-driven-development-using-ruby-part-1.html