RSpec

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

About the Tutorial

RSpec is a unit test framework for the Ruby programming language. RSpec is different than traditional xUnit frameworks like JUnit because RSpec is a Behavior driven development tool. What this means is that, tests written in RSpec focus on the "behavior" of an application being tested. RSpec does not put emphasis on, how the application works but instead on how it behaves, in other words, what the application actually does.

This tutorial will show you, how to use RSpec to test your code when building applications with Ruby.

Audience

This tutorial is for beginners who want to learn how to write better code in Ruby. After finishing this tutorial, you will be able to incorporate RSpec tests into your daily coding practices.

Prerequisites

In order to benefit from reading this tutorial, you should have some experience with programming, specifically with Ruby.

Disclaimer & Copyright

Table of Contents

1. RSPEC – INTRODUCTION

RSpec is a unit test framework for the Ruby programming language. RSpec is different than traditional xUnit frameworks like JUnit because RSpec is a Behavior driven development tool. What this means is that, tests written in RSpec focus on the "behavior" of an application being tested. RSpec does not put emphasis on, how the application works but instead on how it behaves, in other words, what the application actually does.

RSpec Environment

First of all, you will need to install Ruby on your computer. However, if you haven't already done earlier, then you can download and install Ruby from the main Ruby website: https://www.ruby-lang.org/en/documentation/installation.

If you are installing Ruby on Windows, you should have the Ruby installer for Windows here at: http://www.rubyinstaller.org

For this tutorial, you will only need text editor, such as Notepad and a command line console. The examples here will use cmd.exe on Windows.

To run cmd.exe, simply click on the Start menu and type "cmd.exe", then hit the Return key.

At the command prompt in your cmd.exe window, type the following command to see what version of Ruby you are using:

```
ruby -v
```

You should see the below output that looks similar to this:

```
ruby 2.2.3p173 (2015-08-18 revision 51636) [x64-mingw32]
```

The examples in this tutorial will use Ruby 2.2.3 but any version of Ruby higher than 2.0.0 will suffice. Next, we need to install the RSpec gem for your Ruby installation. A gem is a Ruby library which you can use in your own code. In order to install a gem, you need to use the **gem** command.

Let's install the Rspec gem now. Go back to your cmd.exe Window and type the following:

```
gem install rspec
```

You should have a list of dependent gems that were installed, these are gems that the rspec gem needs to function correctly. At the end of the output, you should see something that looks like this:

```
Done installing documentation for diff-lcs, rspec-support, rspec-mocks, rspec-
expectations, rspec-core, rspec after 22 seconds

6 gems installed
```

Do not worry, if your output does not look exactly the same. Also, if you are using a Mac or Linux computer, you may need to either run **gem install rspec** command using **sudo** or use a tool like HomeBrew or RVM to install the rspec gem.

```
Hello World
```

To get started, let's create a directory (folder) to store our RSpec files. In your cmd.exe window, type the following:

```
cd \
```

Then type:

```
mkdir rspec_tutorial
```

And finally, type:

```
cd rspec_tutorial
```

From here, we're going to create another directory named spec, do that by typing:

```
mkdir spec
```

We are going to store our RSpec files in this folder. RSpec files are known as "specs". If this seems confusing to you, you can think of a spec file as a test file. RSpec uses the term "spec" which is a short form for "specification".

Since, RSpec is a BDD test tool, the goal is to focus on what the application does and whether or not it follows a specification. In behavior driven development, the specification is often described in terms of a "User Story". RSpec is designed to make it clear whether the target code is behaving correctly, in other words following the specification.

Let's return to our Hello World code. Open a text editor and add the following code:

```
class HelloWorld
def say_hello
            "Hello World!"
     end
end


describe HelloWorld do
    context "When testing the HelloWorld class" do
     it "should say 'Hello World' when we call the say_hello method" do
            hw = HelloWorld.new
            message = hw.say_hello
            expect(message).to eq "Hello World!"
```

```
    end
  end
end
```

Next, save this to a file named hello_world_spec.rb in the spec folder that you created above. Now back in your cmd.exe window, run this command:

```
rspec spec spec\hello_world_spec.rb
```

When the command completes, you should see output that looks like this:

```
Finished in 0.002 seconds (files took 0.11101 seconds to load)
1 example, 0 failures
```

Congratulations, you just created and ran your first RSpec unit test!

In the next section, we will continue to discuss the syntax of RSpec files.

2. RSPEC – BASIC SYNTAX

Let's take a closer look at the code of our **HelloWorld** example. First of all, in case it isn't clear, we are testing the functionality of the **HelloWorld** class. This of course, is a very simple class that contains only one method **say_hello**().

Here is the RSpec code again:

```
describe HelloWorld do
    context "When testing the HelloWorld class" do
     it "The say_hello method should return 'Hello World'" do
            hw = HelloWorld.new
            message = hw.say_hello
            expect(message).to eq "Hello World!"
     end
    end
end
```

The describe Keyword

The word **describe** is an RSpec keyword. It is used to define an "Example Group". You can think of an "Example Group" as a collection of tests. The **describe** keyword can take a class name and/or string argument. You also need to pass a block argument to **describe**, this will contain the individual tests, or as they are known in RSpec, the "Examples". The block is just a Ruby block designated by the Ruby **do/end** keywords

The context Keyword

The **context** keyword is similar to **describe**. It too can accept a class name and/or string argument. You should use a block with **context** as well. The idea of context is that it encloses tests of a certain type.

For example, you can specify groups of Examples with different contexts like this:

```
context "When passing bad parameters to the foobar() method"
context "When passing valid parameters to the foobar() method"
context "When testing corner cases with the foobar() method"
```

The **context** keyword is not mandatory, but it helps to add more details about the examples that it contains.

The it Keyword

The word **it** is another RSpec keyword which is used to define an "Example". An example is basically a test or a test case. Again, like **describe** and **context, it** accepts both class name and string arguments and should be used with a block argument, designated with **do/end.** In the case of **it,** it is customary to only pass a string and block argument. The string argument often uses the word "should" and is meant to describe what specific behavior should happen inside the **it block**. In other words, it describes that expected outcome is for the Example.

Note the **it block** from our HelloWorld Example:

```
it "The say_hello method should return 'Hello World'" do
```

The string makes it clear what should happen when we call say hello on an instance of the HelloWorld class. This part of the RSpec philosophy, an Example is not just a test, it's also a specification (a spec). In other words, an Example both documents and tests the expected behavior of your Ruby code.

The expect Keyword

The **expect** keyword is used to define an "Expectation" in RSpec. This is a verification step where we check, that a specific expected condition has been met.

From our HelloWorld Example, we have:

```
expect(message).to eql "Hello World!"
```

The idea with **expect** statements is that they read like normal English. You can say this aloud as "Expect the variable message to equal the string 'Hello World'". The idea is that its descriptive and also easy to read, even for non-technical stakeholders such as project managers.

```
The to keyword
```

The **to** keyword is used as part of **expect** statements. Note that you can also use the **not_to** keyword to express the opposite, when you want the Expectation to be false. You can see that to is used with a dot, **expect(message).to,** because it actually just a regular Ruby method. In fact, all of the RSpec keywords are really just Ruby methods.

```
The eql keyword
```

The **eql** keyword is a special RSpec keyword called a Matcher. You use Matchers to specify what type of condition you are testing to be true (or false).

In our HelloWorld **expect** statement, it is clear that **eql** means string equality. Note that, there are different types of equality operators in Ruby and consequently different corresponding Matchers in RSpec. We will explore the many different types of Matchers in a later section.

3. RSPEC – WRITING SPECS

In this chapter, we will create a new Ruby class, save it in its own file and create a separate spec file to test this class.

First, in our new class, it is called **StringAnalyzer**. It's a simple class that, you guessed it, analyzes strings. Our class has only one method **has_vowels?** which as its names suggests, returns true if a string contains vowels and false if it doesn't. Here's the implementation for **StringAnalyzer**:

```
class StringAnalyzer
      def has_vowels?(str)
            !!(str =~ /[aeio]+/i)
      end
end
```

If you followed the HelloWorld section, you created a folder called C:\rspec_tutorial\spec.

Delete the hello_world.rb file if you have it and save the StringAnalyzer code above to a file called string_analyzer.rb in the C:\rspec_tutorial\spec folder.

Here is the source for our spec file to test StringAnalyzer:

```
require 'string_analyzer'
describe StringAnalyzer do
      context "With valid input" do
            it "should detect when a string contains vowels" do
                  sa = StringAnalyzer.new
                  test_string = 'uuu'
                  expect(sa.has_vowels? test_string).to be true
            end

            it "should detect when a string doesn't contain vowels" do
                  sa = StringAnalyzer.new
                  test_string = 'bcdfg'
                  expect(sa.has_vowels? test_string).to be false
            end
      end
end
```

Save this in the same spec directory, giving it the name string_analyzer_test.rb.

In your cmd.exe window, cd to the C:\rspec_tutorial folder and run this command: dir spec

You should see the following:

Directory of C:\rspec_tutorial\spec

```
09/13/2015  08:22 AM    <DIR>          .
09/13/2015  08:22 AM    <DIR>          ..
09/12/2015  11:44 PM                81 string_analyzer.rb
09/12/2015  11:46 PM               451 string_analyzer_test.rb
```

Now we're going to run our tests, run this command: rspec spec

When you pass the name of a folder to **rspec**, it runs all of the spec files inside of the folder. You should see this result:

```
No examples found.


Finished in 0 seconds (files took 0.068 seconds to load)
0 examples, 0 failures
```

The reason that this happened is that, by default, **rspec** only runs files whose names end in "_spec.rb". Rename string_analyzer_test.rb to string_analyzer_spec.rb. You can do that easily by running this command:

```
ren spec\string_analyzer_test.rb string_analyzer_spec.rb
```

Now, run **rspec** spec again, you should see output that looks like this:

```
F.
Failures:


  1) StringAnalyzer With valid input should detect when a string contains
vowels
     Failure/Error: expect(sa.has_vowels? test_string).to be true
       expected true
            got false
     # ./spec/string_analyzer_spec.rb:9:in `block (3 levels) in <top
(required)>'


Finished in 0.015 seconds (files took 0.12201 seconds to load)
```

```
2 examples, 1 failure

Failed examples:

rspec ./spec/string_analyzer_spec.rb:6 # StringAnalyzer With valid input should
detect when a string contains vowels

Do you see what just happened?  Our spec failed because we have a bug in
StringAnalyzer.  The bug is simple to fix, open up string_analyzer.rb in a text
editor and change this line:

!!(str =~ /[aeio]+/i)

to this:

!!(str =~ /[aeiou]+/i)
```

Now, save the changes you just made in string_analyizer.rb and run the rspec spec command again, you should now see output that looks like:

```
..

Finished in 0.002 seconds (files took 0.11401 seconds to load)

2 examples, 0 failures
```

Congratulations, the examples (tests) in your spec file are now passing. We fixed a bug in the regular expression which has vowels method but our tests are far from complete.

It would make sense to add more examples that tests various types of input strings with the has vowels method.

The following table shows some of the permutations that could be added in new Examples (it blocks)

Input string	Description	Expected result with has_vowels?
'aaa', 'eee', 'iii', 'o'	Only one vowel and no other letters.	true
'abcefg'	'At least one vowel and some consonants'	true
'mnklp'	Only consonants.	false
''	Empty string (no letters)	false
'abcde55345&??'	Vowels, consonants, numbers and punctuation characters.	true

'423432%%%^&'	Numbers and punctuation characters only.	false
'AEIOU'	Upper case vowels only.	true
'AeiOuuuA'	Upper case and lower vowels only.	true
'AbCdEfghI'	Upper and lower case vowels and consonants.	true
'BCDFG'	Upper case consonants only.	false
' '	Whitespace characters only.	false

It is up to you to decide, which examples to add to your spec file. There are many conditions to test for, you need to determine what subset of conditions is most important and tests your code the best.

The **rspec** command offers many different options, to see them all, type **rspec** -help. The following table lists the most popular options and describes what they do.

Option/flag	Description
-I PATH	Adds PATH to the load (require) path that **rspec** uses when looking for Ruby source files.
-r, --require PATH	Adds a specific source file to be required in your spec. file(s).
--fail-fast	With this option, rspec will stop running specs after the first Example fails. By default, rspec runs all specified spec files, no matter how many failures there are.
-f, --format FORMATTER	This option allows you to specify different output formats. See the section on Formatters for more details about output formats.
-o, --out FILE	This option directs rspec to write the test results to the output file FILE instead of to standard out.
-c, --color	Enables color in rspec's output. Successful Example results will display in green text, failures will print in red text.
-b, --backtrace	Displays full error backtraces in rspec's output.

-w, --warnings	Displays Ruby warnings in rspec's output.
-P, --pattern PATTERN	Load and run spec files that match the pattern PATTERN. For example, if you pass -p "*.rb", rspec will run all Ruby files, not just the ones that end in "_spec.rb"
-e, --example STRING	This option directs rspec to run all Examples that contain the text STRING in their descriptions.
-t, --tag TAG	With this option, rspec will only run examples that contain the tag TAG. Note that TAG is specified as a Ruby symbol. See the section on RSpec Tags for more details.

4. RSPEC – MATCHERS

If you recall our original Hello World example, it contained a line that looked like this:

```
expect(message).to eq "Hello World!"
```

The keyword eql is an **RSpec** "matcher". Here, we will introduce the other types of matchers in RSpec.

Equality/Identity Matchers

Matchers to test for object or value equality.

Matcher	Description	Example
eq	Passes when actual == expected	expect(actual).to eq expected
eql	Passes when actual.eql?(expected)	expect(actual).to eql expected
be	Passes when actual.equal?(expected)	expect(actual).to be expected
equal	Also passes when actual.equal?(expected)	expect(actual).to equal expected

Example

```
describe "An example of the equality Matchers" do
            it "should show how the equality Matchers work" do
                  a = "test string"
                  b = a
                  # The following Expectations will all pass
expect(a).to eq "test string"
                  expect(a).to eql "test string"
                  expect(a).to be b
                  expect(a).to equal b
            end
end
```

When the above code is executed, it will produce the following output. The number of seconds may be slightly different on your computer:

```
.
Finished in 0.036 seconds (files took 0.11901 seconds to load)
1 example, 0 failures
```

Comparison Matchers

Matchers for comparing to values.

Matcher	Description	Example
>	Passes when actual > expected	expect(actual).to be > expected
>=	Passes when actual >= expected	expect(actual).to be >= expected
<	Passes when actual < expected	expect(actual).to be < expected
<=	Passes when actual <= expected	expect(actual).to be <= expected
be_between inclusive	Passes when actual is <= min and >= max	expect(actual).to be_between(min, max).inclusive
be_between exclusive	Passes when actual is < min and > max	expect(actual).to be_between(min, max).exclusive
match	Passes when actual matches a regular expression	expect(actual).to match(/regex/)

Example

```
describe "An example of the comparison Matchers" do
            it "should show how the comparison Matchers work" do
                  a = 1
                  b = 2
                  c = 3
```

```
                    d = 'test string'
                    # The following Expectations will all pass
expect(b).to be > a
                    expect(a).to be >= a
                    expect(a).to be < b
                    expect(b).to be <= b
                    expect(c).to be_between(1,3).inclusive
                    expect(b).to be_between(1,3).exclusive
                    expect(d).to match /TEST/i
            end
end
```

When the above code is executed, it will produce the following output. The number of seconds may be slightly different on your computer:

```
.
Finished in 0.013 seconds (files took 0.11801 seconds to load)
1 example, 0 failures
```

Class/Type Matchers

Matchers for testing the type or class of objects.

Matcher	Description	Example
be_instance_of	Passes when actual is an instance of the expected class.	expect(actual).to be_instance_of(Expected)
be_kind_of	Passes when actual is an instance of the expected class or any of its parent classes.	expect(actual).to be_kind_of(Expected)
respond_to	Passes when actual responds to the specified method.	expect(actual).to respond_to(expected)

Example

```
describe "An example of the type/class Matchers" do
            it "should show how the type/class Matchers work" do
```

```
                x = 1
                y = 3.14
                z = 'test string'
                # The following Expectations will all pass
                expect(x).to be_instance_of Fixnum
                expect(y).to be_kind_of Numeric
                expect(z).to respond_to(:length)
        end
end
```

When the above code is executed, it will produce the following output. The number of seconds may be slightly different on your computer:

```
.
Finished in 0.002 seconds (files took 0.12201 seconds to load)
1 example, 0 failures
```

True/False/Nil Matchers

Matchers for testing whether a value is true, false or nil.

Matcher	Description	Example
be true	Passes when actual == true	expect(actual).to be true
be false	Passes when actual == false	expect(actual).to be false
be_truthy	Passes when actual is not false or nil	expect(actual).to be_truthy
be_falsey	Passes when actual is false or nil	expect(actual).to be_falsey
be_nil	Passes when actual is nil	expect(actual).to be_nil

Example

```
describe "An example of the true/false/nil Matchers" do
        it "should show how the true/false/nil Matchers work" do
                x = true
```

```
                y = false
                z = nil
                a = "test string"
                # The following Expectations will all pass
                expect(x).to be true
                expect(y).to be false
                expect(a).to be_truthy
                expect(z).to be_falsey
                expect(z).to be_nil
            end
end
```

When the above code is executed, it will produce the following output. The number of seconds may be slightly different on your computer:

```
.
Finished in 0.003 seconds (files took 0.12301 seconds to load)
1 example, 0 failures
```

Error Matchers

Matchers for testing, when a block of code raises an error.

Matcher	Description	Example
raise_error(ErrorClass)	Passes when the block raises an error of type ErrorClass.	expect {block}.to raise_error(ErrorClass)
raise_error("error message")	Passes when the block raise an error with the message "error message".	expect {block}.to raise_error("error message")
raise_error(ErrorClass, "error message")	Passes when the block raises an error of type ErrorClass with the message "error message"	expect {block}.to raise_error(ErrorClass,"error message")

Example

Save the following code to a file with the name **error_matcher_spec.rb** and run it with this command: **rspec error_matcher_spec.rb**.

```
describe "An example of the error Matchers" do
            it "should show how the error Matchers work" do
                  # The following Expectations will all pass
                  expect { 1/0 }.to raise_error(ZeroDivisionError)
                  expect { 1/0 }.to raise_error("divided by 0")
                  expect { 1/0 }.to raise_error("divided by 0",
ZeroDivisionError)
            end
end
```

When the above code is executed, it will produce the following output. The number of seconds may be slightly different on your computer:

```
.
Finished in 0.002 seconds (files took 0.12101 seconds to load)
1 example, 0 failures
```

5. RSPEC – TEST DOUBLES

In this chapter, we will discuss RSpec Doubles, also known as RSpec Mocks. A Double is an object which can "stand in" for another object. You're probably wondering what that means exactly and why you'd need one.

Let's say you are building an application for a school and you have a class representing a classroom of students and another class for students, that is you have a Classroom class and a Student class. You need to write the code for one of the classes first, so let's say that, start with the Classroom class:

```
class ClassRoom
      def initialize(students)
            @students = students
      end

      def list_student_names
            @students.map(&:name).join(',')
      end
end
```

This is a simple class, it has one method list_student_names, which returns a comma delimited string of student names. Now, we want to create tests for this class but how do we do that if we haven't created the Student class yet? We need a test Double.

Also, if we have a "dummy" class that behaves like a Student object then our ClassRoom tests will not depend on the Student class. We call this test isolation.

If our ClassRoom tests don't rely on any other classes, then when a test fails, we can know immediately that there is a bug in our ClassRoom class and not some other class. Keep in mind that, in the real world, you may be building a class that needs to interact with another class written by someone else.

This is where RSpec Doubles (mocks) become useful. Our list_student_names method calls the name method on each Student object in its @students member variable. Therefore, we need a Double which implements a name method.

Here is the code for ClassRoom along with an RSpec Example (test), yet notice that there is no Student class defined:

```
class ClassRoom
      def initialize(students)
            @students = students
      end
```

```
      def list_student_names
            @students.map(&:name).join(',')
      end
end

describe ClassRoom do
it 'the list_student_names method should work correctly' do
            student1 = double('student')
            student2 = double('student')
            allow(student1).to receive(:name) { 'John Smith'}
            allow(student2).to receive(:name) { 'Jill Smith'}

            cr = ClassRoom.new [student1,student2]

            expect(cr.list_student_names).to eq('John Smith,Jill Smith')
      end
end
```

When the above code is executed, it will produce the following output. The elapsed time may be slightly different on your computer:

```
.
Finished in 0.01 seconds (files took 0.11201 seconds to load)
1 example, 0 failures
```

As you can see, using a **test double** allows you to test your code even when it relies on a class that is undefined or unavailable. Also, this means that when there is a test failure, you can tell right away that it's because of an issue in your class and not a class written by someone else.

6. RSPEC – STUBS

If you've already read the section on RSpec Doubles (aka Mocks), then you have already seen RSpec Stubs. In RSpec, a stub is often called a Method Stub, it's a special type of method that "stands in" for an existing method, or for a method that doesn't even exist yet.

Here is the code from the section on RSpec Doubles:

```
class ClassRoom
      def initialize(students)
            @students = students
      End

      def list_student_names
            @students.map(&:name).join(',')
      end
end
describe ClassRoom do
it 'the list_student_names method should work correctly' do
            student1 = double('student')
            student2 = double('student')
            allow(student1).to receive(:name) { 'John Smith'}
            allow(student2).to receive(:name) { 'Jill Smith'}
            cr = ClassRoom.new [student1,student2]
            expect(cr.list_student_names).to eq('John Smith,Jill Smith')
      end
end
```

In our example, the allow() method provides the method stubs that we need to test the ClassRoom class. In this case, we need an object that will act just like an instance of the Student class, but that class doesn't actually exist (yet). We know that the Student class needs to provide a name() method and we use allow() to create a method stub for name ().

One thing to note is that, RSpec's syntax has changed a bit over the years. In older versions of RSpec, the above method stubs would be defined like this:

```
student1.stub(:name).and_return('John Smith')
student2.stub(:name).and_return('Jill Smith')
```

Let's take the above code and replace the two **allow()** lines with the old RSpec syntax:

```
class ClassRoom
     def initialize(students)
           @students = students
     end
     def list_student_names
           @students.map(&:name).join(',')
     end
end
describe ClassRoom do
it 'the list_student_names method should work correctly' do
           student1 = double('student')
           student2 = double('student')
           student1.stub(:name).and_return('John Smith')
student2.stub(:name).and_return('Jill Smith')
           cr = ClassRoom.new [student1,student2]
           expect(cr.list_student_names).to eq('John Smith,Jill Smith')
     end
end
```

You will see this output when you execute the above code:

```
.


Deprecation Warnings:


Using `stub` from rspec-mocks' old `:should` syntax without explicitly enabling
the syntax is deprec
ated. Use the new `:expect` syntax or explicitly enable `:should` instead.
Called from C:/rspec_tuto
rial/spec/double_spec.rb:15:in `block (2 levels) in <top (required)>'.


If you need more of the backtrace for any of these deprecations to
identify where to make the necessary changes, you can configure
```

```
`config.raise_errors_for_deprecations!`, and it will turn the
deprecation warnings into errors, giving you the full backtrace.


1 deprecation warning total

Finished in 0.002 seconds (files took 0.11401 seconds to load)
1 example, 0 failures
```

It's recommended that you use the new allow() syntax when you need to create method stubs in your RSpec examples, but we've provided the older style here so that you will recognize it if you see it.

7. RSPEC – HOOKS

When you are writing unit tests, it is often convenient to run setup and teardown code before and after your tests. Setup code is the code that configures or "sets up" conditions for a test. Teardown code does the cleanup, it makes sure that the environment is in a consistent state for subsequent tests.

Generally speaking, your tests should be independent of each other. When you run an entire suite of tests and one of them fails, you want to have confidence that it failed because the code that it is testing has a bug, not because the previous test left the environment in an inconsistent state.

The most common hooks used in RSpec are before and after hooks. They provide a way to define and run the setup and teardown code we discussed above. Let's consider this example code:

```
class SimpleClass
      attr_accessor :message
      def initialize()
            puts "\nCreating a new instance of the SimpleClass class"
            @message = 'howdy'
      end
      def update_message(new_message)
            @message = new_message
      end
end
describe SimpleClass do
      before(:each) do
            @simple_class = SimpleClass.new
      end
      it 'should have an initial message' do
            expect(@simple_class).to_not be_nil
            @simple_class.message = 'Something else. . .'
      end
      it 'should be able to change its message' do
            @simple_class.update_message('a new message')
            expect(@simple_class.message).to_not be 'howdy'
      end
end
```

When you run this code, you'll get the following output:

```
Creating a new instance of the SimpleClass class
.
Creating a new instance of the SimpleClass class
.
Finished in 0.003 seconds (files took 0.11401 seconds to load)
2 examples, 0 failures
```

Let's take a closer look at what's happening. The before(:each) method is where we define the setup code. When you pass the :each argument, you are instructing the before method to run before each example in your Example Group i.e. the two it blocks inside the describe block in the code above.

In the line: @simple_class = SimpleClass.new, we are creating a new instance of the SimpleClass class and assigning it to an instance variable of an object. What object you might be wondering? RSpec creates a special class behind the scenes in the scope of the describe block. This allows you to assign values to instance variables of this class, that you can access within the it blocks in your Examples. This also makes it easy to write cleaner code in our tests. If each test (Example) needs an instance of SimpleClass, we can put that code in the before hook and not have to add it to each example.

Notice that, the line "Creating a new instance of the SimpleClass class" is written to the console twice, this shows that, before hook was called in each of the **it blocks**.

As we've mentioned, RSpec also has an after hook and both the before and after hooks can take: all as an argument. The after hook will run after the specified target. The: all target means that the hook will run before/after all of the Examples. Here is a simple example that illustrates when each hook is called.

```
describe "Before and after hooks" do
      before(:each) do
            puts "Runs before each Example"
      end
      after(:each) do
            puts "Runs after each Example"
      end
      before(:all) do
            puts "Runs before all Examples"
      end
      after(:all) do
            puts "Runs after all Examples"
      end
      it 'is the first Example in this spec file' do
```

```
            puts 'Running the first Example'
      end
      it 'is the second Example in this spec file' do
            puts 'Running the second Example'
      end
end
```

When you run the above code, you will see this output:

```
Runs before all Examples
Runs before each Example
Running the first Example
Runs after each Example
.Runs before each Example
Running the second Example
Runs after each Example
.Runs after all Examples
```

8. RSPEC – TAGS

RSpec Tags provide an easy way to run specific tests in your spec files. By default, RSpec will run all tests in the spec files that it runs, but you might only need to run a subset of them. Let's say that you have some tests that run very quickly and that you've just made a change to your application code and you want to just run the quick tests, this code will demonstrate how to do that with RSpec Tags.

```
describe "How to run specific Examples with Tags" do
      it 'is a slow test', :slow => true do
            sleep 10
            puts 'This test is slow!'
      end
      it 'is a fast test', :fast => true do
            puts 'This test is fast!'
      end
end
```

Now, save the above code in a new file called tag_spec.rb. From the command line, run this command: rspec --tag slow tag_spec.rb

You will see this output:

Run options: include {: slow=>true}

```
This test is slow!
.
Finished in 10 seconds (files took 0.11601 seconds to load)
1 example, 0 failures
```

Then, run this command: rspec --tag fast tag_spec.rb

You will see this output:

```
Run options: include {:fast=>true}
This test is fast!
.
Finished in 0.001 seconds (files took 0.11201 seconds to load)
1 example, 0 failures
```

As you can see, RSpec Tags makes it very easy to a subset of tests!

9. RSPEC – SUBJECTS

One of RSpec's strengths is that it provides many ways to write tests, clean tests. When your tests are short and uncluttered, it becomes easier to focus on the expected behavior and not on the details of how the tests are written. RSpec Subjects are yet another shortcut allowing you to write simple straightforward tests.

Consider this code:

```
class Person
      attr_reader :first_name, :last_name
      def initialize(first_name, last_name)
            @first_name = first_name
            @last_name = last_name
      end
end
describe Person do
      it 'create a new person with a first and last name' do
            person = Person.new 'John', 'Smith'
            expect(person).to have_attributes(first_name: 'John')
            expect(person).to have_attributes(last_name: 'Smith')
      end
end
```

It's actually pretty clear as is, but we could use RSpec's subject feature to reduce the amount of code in the example. We do that by moving the person object instantiation into the describe line.

```
class Person
      attr_reader :first_name, :last_name
      def initialize(first_name, last_name)
            @first_name = first_name
            @last_name = last_name
      end
end
describe Person.new 'John', 'Smith' do
      it { is_expected.to have_attributes(first_name: 'John') }
      it { is_expected.to have_attributes(last_name: 'Smith') }
```

```
end
```

When you run this code, you will see this output:

```
..
Finished in 0.003 seconds (files took 0.11201 seconds to load)
2 examples, 0 failures
```

Note, how much simpler the second code sample is. We took the one **it block** in the first example and replaced it with two **it blocks** which end up requiring less code and are just as clear.

10. RSPEC – HELPERS

Sometimes your RSpec examples need an easy way to share reusable code. The best way to accomplish this is with Helpers. Helpers are basically regular Ruby methods which you share across examples. To illustrate the benefit of using helpers, let's consider this code:

```
class Dog
      attr_reader :good_dog, :has_been_walked
      def initialize(good_or_not)
            @good_dog = good_or_not
            @has_been_walked = false
      end
      def walk_dog
            @has_been_walked = true
      end
end
describe Dog do
      it 'should be able to create and walk a good dog' do
            dog = Dog.new(true)
            dog.walk_dog
expect(dog.good_dog).to be true
            expect(dog.has_been_walked).to be true
      end
      it 'should be able to create and walk a bad dog' do
            dog = Dog.new(true)
            dog.walk_dog
            expect(dog.good_dog).to be true
            expect(dog.has_been_walked).to be true
      end
end
```

This code is clear, but it's always a good idea to reduce repeated code whenever possible. We can take the above code and reduce some of this repetition with a helper method called create_and_walk_dog().

```
class Dog
      attr_reader :good_dog, :has_been_walked
      def initialize(good_or_not)
            @good_dog = good_or_not
            @has_been_walked = false
      end
      def walk_dog
            @has_been_walked = true
      end
end
describe Dog do
      def create_and_walk_dog(good_or_bad)
            Dog.new(good_or_bad).walk_dog
end
      it 'should be able to create and walk a good dog' do
            dog = create_and_walk_dog(true)
expect(dog.good_dog).to be true
            expect(dog.has_been_walked).to be true
      end
      it 'should be able to create and walk a bad dog' do
            dog = create_and_walk_dog(false)
            expect(dog.good_dog).to be false
            expect(dog.has_been_walked).to be true
      end
end
```

When you run the above code, you will see this output:

```
..
Finished in 0.002 seconds (files took 0.11401 seconds to load)
2 examples, 0 failures
```

As you can see, we were able to push the logic for creating and walking a dog object into a Helper which allows our examples to be shorter and cleaner.

11. RSPEC – METADATA

RSpec is a flexible and powerful tool. The Metadata functionality in RSpec is no exception. Metadata generally refers to "data about data". In RSpec, this means data about your **describe**, **context** and **it blocks**.

Let's take a look at an example:

```
RSpec.describe "An Example Group with a metadata variable", :foo => 17 do
  context 'and a context with another variable', :bar => 12 do
    it 'can access the metadata variable of the outer Example Group' do |example|
            expect(example.metadata[:foo]).to eq(17)
    end

    it 'can access the metadata variable in the context block' do |example|
            expect(example.metadata[:bar]).to eq(12)
    end
end
```

When you run the above code, you will see this output:

```
..
Finished in 0.002 seconds (files took 0.11301 seconds to load)
2 examples, 0 failures
```

Metadata provides a way to assign variables at various scopes within your RSpec files. The example.metadata variable is a Ruby hash which contains other information about your Examples and Example groups.

For instance, let's rewrite the above code to look like this:

```
RSpec.describe "An Example Group with a metadata variable", :foo => 17 do

  context 'and a context with another variable', :bar => 12 do
    it 'can access the metadata variable in the context block' do |example|
            expect(example.metadata[:foo]).to eq(17)
            expect(example.metadata[:bar]).to eq(12)
            example.metadata.each do |k,v|
                    puts "#{k}: #{v}"
            end
```

```
    end
  end
end
```

When we run this code, we see all of the values in the example.metadata hash:

```
.execution_result: #<RSpec::Core::Example::ExecutionResult:0x00000002befd50>

block: #<Proc:0x00000002bf81a8@C:/rspec_tutorial/spec/metadata_spec.rb:7>

description_args: ["can access the metadata variable in the context block"]

description: can access the metadata variable in the context block

full_description: An Example Group with a metadata variable and a context with
another variable can access the metadata variable in the context block

described_class:

file_path: ./metadata_spec.rb

line_number: 7

location: ./metadata_spec.rb:7

absolute_file_path: C:/rspec_tutorial/spec/metadata_spec.rb

rerun_file_path: ./metadata_spec.rb

scoped_id: 1:1:2

foo: 17

bar: 12

example_group:
{:execution_result=>#<RSpec::Core::Example::ExecutionResult:0x00000002bfa0e8>,
:block=>#<Proc:0x00000002bfac00@C:/rspec_tutorial/spec/metadata_spec.rb:2>, :de
scription_args=>["and a context with another variable"], :description=>"and a
context with another variable", :full_description=>"An Example Group with a
metadata variable and a context with another
variable", :described_class=>nil, :file_path=>"./metadata_spec.rb", :line_numbe
r=>2, :location=>"./metadata_spec.rb:2", :absolute_file_path=>"C:/rspec_tutoria
l/spec/metadata_spec.rb", :rerun_file_path=>"./metadata_spec.rb", :scoped_id=>"
1:1", :foo=>17, :parent_example_group=>{:execution_result=>#<RSpec::Core::Examp
le::ExecutionResult:0x00000002c1f690>, :block=>#<Proc:0x00000002baff70@C:/rspec
_tutorial/spec/metadata_spec.rb:1>, :description_args=>["An Example Group with
a metadata variable"], :description=>"An Example Group with a metadata
variable", :full_description=>"An Example Group with a metadata
variable", :described_class=>nil, :file_path=>"./metadata_spec.rb", :line_numbe
r=>1, :location=>"./metadata_spec.rb:1",:absolute_file_path=>"C:/rspec_tutorial
/spec/metadata_spec.rb", :rerun_file_path=>"./metadata_spec.rb", :scoped_id=>"1
", :foo=>17}, :bar=>12}shared_group_inclusion_backtrace: []

last_run_status: unknown

.

Finished in 0.004 seconds (files took 0.11101 seconds to load)
```

```
2 examples, 0 failures
```

Most likely, you will not need to use all of this metadata, but look at the full description value:

An Example Group with a metadata variable and a context with another variable can access the metadata variable in the context block

This is a sentence created from the describe block description + its contained context block description + the description for the **it block**.

What is interesting to note here is that, these three strings together read like a normal English sentence. . . which is one of the ideas behind RSpec, having tests that sound like English descriptions of behavior.

12. RSPEC – FILTERING

You may want to read the section on RSpec Metadata before reading this section because, as it turns out, RSpec filtering is based on RSpec Metadata.

Imagine that you have a spec file and it contains two types of tests (Examples): positive functional tests and negative (error) tests. Let's define them like this:

```
RSpec.describe "An Example Group with positive and negative Examples" do
      context 'when testing Ruby\'s build-in math library' do
            it 'can do normal numeric operations' do
                  expect(1 + 1).to eq(2)
            end
            it 'generates an error when expected' do
                  expect{1/0}.to raise_error(ZeroDivisionError)
            end
      end
end
```

Now, save the above text as a file called 'filter_spec.rb' and then run it with this command:

```
rspec filter_spec.rb
```

You will see output that looks something like this:

```
..
Finished in 0.003 seconds (files took 0.11201 seconds to load)
2 examples, 0 failures
```

Now what if, we wanted to re-run only the positive tests in this file? Or only the negative tests? We can easily do that with RSpec Filters. Change the above code to this:

```
RSpec.describe "An Example Group with positive and negative Examples" do
      context 'when testing Ruby\'s build-in math library' do
            it 'can do normal numeric operations', positive: true do
                  expect(1 + 1).to eq(2)
            end
            it 'generates an error when expected', negative: true do
                  expect{1/0}.to raise_error(ZeroDivisionError)
            end
```

```
    end
end
```

Save your changes to filter_spec.rb and run this slightly different command:

```
rspec --tag positive filter_spec.rb
```

Now, you will see output that looks like this:

```
Run options: include {:positive=>true}
.
Finished in 0.001 seconds (files took 0.11401 seconds to load)
1 example, 0 failures
```

By specifying --tag positive, we're telling RSpec to only run Examples with the: positive metadata variable defined. We could do the same thing with negative tests by running the command like this:

```
rspec --tag negative filter_spec.rb
```

Keep in mind that these are just examples, you can specify a filter with any name that you want.

RSpec Formatters

Formatters allow RSpec to display the output from tests in different ways. Let's create a new RSpec file containing this code:

```
RSpec.describe "A spec file to demonstrate how RSpec Formatters work" do
    context 'when running some tests' do
        it 'the test usually calls the expect() method at least once' do
            expect(1 + 1).to eq(2)
        end
    end
end
```

Now, save this to a file called formatter_spec.rb and run this RSpec command:

```
rspec formatter_spec.rb
```

You should see output that looks like this:

```
.
Finished in 0.002 seconds (files took 0.11401 seconds to load)
1 example, 0 failures
```

Now run the same command but this time specify a formatter, like this:

```
rspec --format progress formatter_spec.rb
```

You should see the same output this time:

```
.

Finished in 0.002 seconds (files took 0.11401 seconds to load)

1 example, 0 failures
```

The reason is that the "progress" formatter is the default formatter. Let's try a different formatter next, try running this command:

```
rspec --format doc formatter_spec.rb
```

Now you should see this output:

```
A spec file to demonstrate how RSpec Formatters work

  when running some tests

    the test usually calls the expect() method at least once


Finished in 0.002 seconds (files took 0.11401 seconds to load)

1 example, 0 failures
```

As you can see, the output is quite different with the "doc" formatter. This formatter presents the output in a documentation-like style. You might be wondering what these options look like when you have a failure in a test (Example). Let's change the code in **formatter_spec.rb** to look like this:

```
RSpec.describe "A spec file to demonstrate how RSpec Formatters work" do

     context 'when running some tests' do

            it 'the test usually calls the expect() method at least once' do

                   expect(1 + 1).to eq(1)

            end

     end

end
```

The expectation **expect(1 + 1).to eq(1)** should fail. Save your changes and re-run the above commands:

rspec --format progress formatter_spec.rb and remember, since the "progress" formatter is the default, you could just run: **rspec formatter_spec.rb**. You should see this output:

```
F

Failures:

  1) A spec file to demonstrate how RSpec Formatters work when running some
tests the test usually c

alls the expect() method at least once

     Failure/Error: expect(1 + 1).to eq(1)


       expected: 1

            got: 2


       (compared using ==)

     # ./formatter_spec.rb:4:in `block (3 levels) in <top (required)>'


Finished in 0.016 seconds (files took 0.11201 seconds to load)

1 example, 1 failure

Failed examples:


rspec ./formatter_spec.rb:3 # A spec file to demonstrate how RSpec Formatters
work when running some tests the test usually calls the expect() method at
least once
```

Now, let's try the doc formatter, run this command:

```
rspec --format doc formatter_spec.rb
```

Now, with the failed test, you should see this output:

```
A spec file to demonstrate how RSpec Formatters work

  when running some tests

    the test usually calls the expect() method at least once (FAILED - 1)


Failures:


  1) A spec file to demonstrate how RSpec Formatters work when running some
tests the test usually calls the expect() method at least once

     Failure/Error: expect(1 + 1).to eq(1)


       expected: 1

            got: 2
```

```
       (compared using ==)
     # ./formatter_spec.rb:4:in `block (3 levels) in <top (required)>'

Finished in 0.015 seconds (files took 0.11401 seconds to load)
1 example, 1 failure
```

Failed Examples

rspec ./formatter_spec.rb:3 # A spec file to demonstrate how RSpec Formatters work when running some tests the test usually calls the expect() method at least once

RSpec Formatters offer the ability to change the way test results display, it is even possible to create your own custom Formatter, but that is a more advanced topic.

13. RSPEC – EXPECTATIONS

When you learn RSpec, you may read a lot about expectations and it can be a bit confusing at first. There are two main details you should keep in mind when you see the term Expectation:

- An Expectation is simply a statement in an **it block** that uses the **expect()** method. That's it. It's no more complicated than that. When you have code like this: **expect(1 + 1).to eq(2)**, you have an Expectation in your example. You are expecting that the expression **1 + 1** evaluates to **2**. The wording is important though since RSpec is a BDD test framework. By calling this statement an Expectation, it is clear that your RSpec code is describing the "behavior" of the code it's testing. The idea is that you are expressing how the code should behave, in a way that reads like documentation.

- The Expectation syntax is relatively new. Before the **expect()** method was introduced (back in 2012), RSpec used a different syntax that was based on the **should()** method. The above Expectation is written like this in the old syntax: **(1 + 1).should eq(2)**

You may encounter the old RSpec syntax for Expectations when working with an older code based or an older version of RSpec. If you use the old syntax with a new version of RSpec, you will see a warning.

For example, with this code:

```
RSpec.describe "An RSpec file that uses the old syntax" do
     it 'you should see a warning when you run this Example' do
            (1 + 1).should eq(2)
     end
end
```

When you run it, you will get an output that looks like this:

```
.
Deprecation Warnings:


Using `should` from rspec-expectations' old `:should` syntax without explicitly
enabling the syntax
is deprecated. Use the new `:expect` syntax or explicitly enable `:should` with
`config.expect_with(
:rspec) { |c| c.syntax = :should }` instead. Called from
C:/rspec_tutorial/spec/old_expectation.rb:3
:in `block (2 levels) in <top (required)>'.
```

```
If you need more of the backtrace for any of these deprecations to
identify where to make the necessary changes, you can configure
`config.raise_errors_for_deprecations!`, and it will turn the
deprecation warnings into errors, giving you the full backtrace.

1 deprecation warning total
Finished in 0.001 seconds (files took 0.11201 seconds to load)
1 example, 0 failures
```

Unless you are required to use the old syntax, it is highly recommended that you use expect() instead of should().