RSpec

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

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.