

**howto**

**Guia Rápido de RSpec**  
por Nando Vieira

Copyright © Simples Ideias & Nando Vieira. Todos os direitos reservados.

Nenhuma parte desta publicação pode ser reproduzida sem o consentimento dos autores.  
Todas as marcas registradas são de propriedade de seus respectivos donos.

Guia Rápido de RSpec, Nando Vieira, 1ª versão

# Conteúdo

- 1 **Introdução**
- 2 **Instalando o RSpec**
- 4 **Configurando o RSpec**
- 8 **Executando exemplos no RSpec**
- 13 **Criando Exemplos**
  - 13 Descrevendo objetos e comportamentos
  - 15 Definindo exemplos na prática
  - 18 Hooks: before, after e around
  - 21 Definindo métodos auxiliares (helpers)
- 26 **RSpec::Expectations**
  - 26 Built-in matchers
  - 35 Criando o seu próprio matcher
  - 40 Definindo o sujeito
- 44 **RSpec::Mocks**
  - 44 Mocks, Doubles e Stubs
  - 45 Method stubbing
  - 48 Message expectation
  - 53 Outros frameworks de mocking
- 55 **Usando RSpec com Ruby on Rails**
  - 55 Configurando o ambiente
  - 56 Models
  - 59 Controllers
  - 72 Views
  - 74 Helpers
  - 76 Requests
- 79 **Outras bibliotecas**
  - 79 Fakeweb
  - 80 FakeFS
  - 81 Delorean
  - 82 Factory Girl
  - 84 Factory Girl Preload
  - 85 SimpleCov
  - 87 Cucumber
  - 92 Steak
  - 93 Shoulda

# RSpec::Mocks

## Mocks, Doubles e Stubs

O RSpec permite criar *mocks*, objetos que simulam serem outros objetos e que podem ter seu comportamento alterado de forma controlada. Eles normalmente são usados em alguns casos específicos:

- quando suas expectativas dependem de objetos complexos
- quando um objeto não fornece resultados determinísticos
- quando um objeto possui estados difíceis de reproduzir
- quando sua execução é lenta
- quando ele ainda não existe

O RSpec permite criar estes objetos com o método `mock()`.

```
describe "User class" do
  it "should return name and email" do
    user = mock(:user, :name => "John Doe", :email => "john@doe.com")

    user.name.should == "John Doe"
    user.email.should == "john@doe.com"
  end
end
```

O método `mock()` recebe um primeiro argumento que irá identificar este mock. Embora este argumento seja opcional, seu uso é recomendado, já que este valor será usado em mensagens de erro. O segundo argumento é um hash que será utilizado como métodos deste objeto.

Este segundo argumento pode ser substituído pelo método `stub()` do próprio objeto.

```

describe "User class" do
  it "should return name and email" do
    user = mock(:user)
    user.stub :name => "John Doe", :email => "john@doe.com"

    user.name.should == "John Doe"
    user.email.should == "john@doe.com"
  end
end

```

Além do método `mock()`, você também pode utilizar os métodos `stub()` e `double()`; não existe nenhuma diferença real entre eles além do nome do método.

Quando você possui um objeto muito complexo, pode querer fazer stubbing de muitos métodos, simplesmente para que eles não lancem uma exceção. Neste caso, você pode permitir que qualquer método seja invocado, sem se preocupar se ele foi definido ou não, retornando o próprio mock.

```

describe "User class" do
  it "should respond to everything" do
    user = mock(:user).as_null_object

    user.name
    user.email
  end
end

```

## Method stubbing

Muitas vezes queremos sobrescrever ou forjar o valor de um método, sem fazer nenhuma expectativa. É aí que entra o método `stub()`.

```

describe User do
  it "should return e-mail" do
    subject.stub :email => "john@doe.com"
  end
end

```

```

    subject.email.should == "john@doe.com"
end

it "should return no e-mail" do
  subject.stub :email
  subject.email.should be_nil
end

it "should override implementation" do
  subject.stub(:email) do
    "john@doe.com"
  end
end

    subject.email.should == "john@doe.com"
end

it "should return several attributes" do
  subject.stub :email => "john@doe.com", :name => "John Doe"

  subject.email.should == "john@doe.com"
  subject.name.should == "John Doe"
end
end

```

Alternativamente, você pode definir o valor de retorno com o método `and_return()`. Se você passar diversos argumentos para o método `and_return()`, cada chamada irá retornar um argumento. Quando não existir mais nenhum argumento, o último é que será sempre retornado.

```

describe User do
  it "should return only one value" do
    subject.stub(:roll_dice).and_return(6)
    subject.roll_dice.should == 6
  end

  it "should return different values for each call" do
    subject.stub(:roll_dice).and_return(6,4,1)
    subject.roll_dice.should == 6
    subject.roll_dice.should == 4
  end
end

```

```
subject.roll_dice.should == 1
end
end
```

## Stub chain

Sempre que você precisar fazer uma chamada encadeada e quiser verificar o último valor, utilize o método `stub_chain`.

```
describe User do
  it "should return the recent things count" do
    subject.stub_chain(:things, :recent, :count => 10)
    subject.things.recent.count.should == 10
  end

  it "should return the older things count" do
    subject.stub_chain(:things, :older, :count).and_return(15)
    subject.things.older.count.should == 15
  end
end
```

## Gerando exceções

Você pode fazer com que um método lance uma exceção caso ele seja invocado. Você pode passar uma string, uma classe ou instância que herde de `Exception`.

```
describe User do
  it "should raise" do
    subject.stub(:age).and_raise("Not implemented")
    expect { subject.age }.to raise_error("Not implemented")
  end

  it "should throw" do
    subject.stub(:destroy).and_throw(:destroy_record)
    expect { subject.destroy }.to throw_symbol(:destroy_record)
  end
end
```

```
end  
end
```

## Executando blocos

Você pode fazer com que uma chamada a um método faça o *yielding* de argumentos.

```
describe User do  
  it "should yield" do  
    User.stub(:create).and_yield(subject)  
  
    User.create do |user|  
      user.should == subject  
    end  
  end  
end
```

## Message expectation

Para criar expectativas, verificando se um método foi executado e com quais argumentos.

```
describe User do  
  it "should return e-mail address" do  
    subject.should_receive(:email).and_return("john@doe.com")  
    subject.should_not_receive(:name)  
  
    subject.email.should == "john@doe.com"  
  end  
end
```

Você também pode criar expectativas garantindo que um determinado método não seja executado com o método `should_not_receive()`.

```
describe User do
  it "should not call #create" do
    User.should_not_receive(:create).never
    User.new
  end
end
```

## Counts

Você pode criar expectativas de quantas vezes um método foi invocado.

```
describe User do
  it "should instantiate 1 user" do
    User.should_receive(:new).once
    User.new
  end

  it "should instantiate 3 users" do
    User.should_receive(:new).exactly(3).times
    3.times { User.new }
  end

  it "should instantiate 2 users" do
    User.should_receive(:new).twice
    2.times { User.new }
  end

  it "should instantiate at most 3 users" do
    User.should_receive(:new).at_most(3).times
    2.times { User.new }
  end

  it "should instantiate at least 2 users" do
    User.should_receive(:new).at_least(2).times
    3.times { User.new }
  end

  it "should instantiate at least 1 user" do
    User.should_receive(:new).at_least(1).times
  end
end
```

```

    3.times { User.new }
  end

  it "should instantiate any number of users" do
    User.should_receive(:new).any_number_of_times
      rand(3).times { User.new }
    end

  it "should never instantiate a user" do
    User.should_receive(:new).never
  end
end

```

## Argument matchers

Além de definir expectativas em um método, você pode especificar quais argumentos este método deve receber quando executado. Para isso, use o método `with()`.

```

describe User do
  it "should initialize user with name" do
    User.should_receive(:new).with(:name => "John Doe")
    User.new :name => "John Doe"
  end
end

```

Além de explicitamente dizer quais argumentos o método deve receber, você também pode fazer algumas expectativas quanto ao tipo e comportamento dos argumentos.

### Com um ou mais argumentos, de qualquer tipo

Especifica se o método recebeu um ou mais argumentos, independente do tipo destes argumentos.

```

it "should initialize with anything" do
  User.should_receive(:new).with(anything)
  User.new(:name => "John Doe")
end

```

## Com qualquer quantidade de argumentos, de qualquer tipo

Especifica se o método foi executado, independente de quantos argumentos foram passados.

```
it "should initialize with or without arguments" do
  User.should_receive(:new).with(any_args).twice
  User.new
  User.new(:name => "John Doe")
end
```

## Com os valores de um hash

Especifica se o método foi executado com um hash contendo determinados valores.

```
it "should initialize with name" do
  User.should_receive(:new).with(hash_including(:name => "John Doe"))
  User.new(:email => "john@doe.com", :name => "John Doe")
end
```

## Sem os valores de um hash

Especifica se o método foi executado com um hash que não contém determinados valores.

```
it "should initialize without name" do
  User.should_receive(:new).with(hash_not_including(:name => "John Doe"))
  User.new(:email => "john@doe.com")
end
```

## Com uma string em um determinado padrão

Especifica se o método foi executado com uma string que faz casa uma expressão regular.

```
it "should set name" do
  subject.should_receive(:name=).with(/John/)
  subject.name = "John Doe"
end
```

## Com uma instância

Especifica se o método foi executado com uma instância de uma determinada classe.

```
it "should set name" do
  subject.should_receive(:name=).with(instance_of(String))
  subject.name = "John Doe"
end
```

Você também pode especificar se ele deve apenas descender de uma superclasse.

```
it "should set name" do
  Object.const_set(:SuperString, Class.new(String))

  subject.should_receive(:name=).with(kind_of(String))
  subject.name = SuperString.new
end
```

## Com um objeto que responde a um método

Especifica se o método foi executado e recebeu um objeto que responde a um determinado método.

```
it "should set name" do
  subject.should_receive(:name=).with(duck_type(:<<)).twice

  subject.name = "John Doe"
  subject.name = ["John Doe"]
end
```

## Com um valor booleano

Especifica se o método foi executado com os valores `true` ou `false`.

```
it "should not be admin" do
  subject.should_receive(:admin=).with(boolean)
  subject.admin = false
end
```

## Ordem das chamadas

Às vezes, a ordem em que os métodos foram executados é muito importante. Neste caso, você deve utilizar o método `ordered()`.

```
describe User do
  it "should call in sequence" do
    User.should_receive(:new).ordered
    User.should_receive(:create).ordered

    User.new
    User.create
  end
end
```

## Outros frameworks de mocking

Embora o RSpec possua um framework de mocks bem completo, você pode preferir usar algum outro. O RSpec suporta os frameworks [Mocha](#), [RR](#) e [Flexmock](#).

Para ativar o suporte a outro framework, basta utilizar o método `RSpec.configure`.

```
RSpec.configure do |config|
  config.mock_framework = :mocha
end
```

Isso irá desativar todos os métodos de mocking do RSpec por completo, dando lugar aos métodos definidos pelo framework que você escolheu. Por exemplo, veja como fica um exemplo utilizando o Mocha.

```
it "should initialize user" do
  User.expects(:new).with(:name => "John Doe").returns(subject)
  User.new :name => "John Doe"
end
```

O mesmo exemplo poderia ser escrito com o framework padrão do RSpec da seguinte forma:

```
it "should initialize user" do
  User.should_receive(:new).with(:name => "John Doe").and_return(subject)
  User.new :name => "John Doe"
end
```

Você também pode passar um módulo. Neste caso, ele deve implementar a API de frameworks de mocking, implementando 3 métodos:

- `setup_mocks_for_rspec()`: executado antes de cada exemplo.
- `verify_mocks_for_rspec()`: executado depois de cada exemplo. Deve lançar uma exceção caso alguma expectativa falhe.
- `teardown_mocks_for_rspec()`: executado depois do método `verify_mocks_for_rspec()`, quando nenhuma expectativa falhar.

