howto

Guia Rápido de RSpec

por Nando Vieira

Guia Rápido de RSpec, Nando Vieira, 1ª versão

Conteúdo

CAPÍTULO 1

Introdução

O RSpec é um framework de testes escrito em Ruby, que permite que você descreva sua aplicação em uma DSL (Domain Specific Language) muito simples e elegante.

Como a maioria dos projetos falha em manter uma documentação atualizada, utilizar o RSpec pode ser uma excelente maneira de se documentar um projeto, já que o resultado dos exemplos executados, quando bem escritos, formam uma excelente especificação do projeto.

O RSpec é muito usado para descrever aplicações Ruby on Rails, mas você pode utilizá-lo para descrever qualquer código escrito em Ruby. Ele é composto por diversos módulos que permitem expressar histórias, cenários e expectativas de como sua aplicação e/ou objetos devem se comportar.

Embora o RSpec seja muito completo, é possível estendê-lo muito facilmente caso você queira adequar o modo como ele funciona. Isso permite tornar os seus testes ainda mais expressivos e concisos.

Neste guia rápido você verá tudo[1] sobre o RSpec, com exemplos de uso e muito código!

– Nando Vieira, 26 de janeiro de 2011

[1] Ou quase tudo que o RSpec oferece. Como eu disse, as funcionalidades são muitas e alguma coisa pode ter ficado de fora.

CAPÍTULO 1

Instalando o RSpec

Parar instalar o RSpec, basta executar o comando `gem install rspec`.

```
$ gem install rspec
Fetching: rspec-core-2.11.1.gem (100%)
Fetching: rspec-expectations-2.11.2.gem (100%)
Fetching: rspec-mocks-2.11.2.gem (100%)
Fetching: rspec-2.11.0.gem (100%)
Successfully installed rspec-core-2.11.1
Successfully installed rspec-expectations-2.11.2
Successfully installed rspec-mocks-2.11.2
Successfully installed rspec-2.11.0
4 gems installed
```

Se você utiliza a gem Bundler, precisará adicionar a dependência ao arquivo `Gemfile`, geralmente localizado na raíz de seu projeto. O Gemfile será parecido com isto:

```
source :rubygems
gem "rspec"
```

Normalmente, seus exemplos[1] ficam localizados no diretório `spec`, com uma estrutura semelhante a esta:

[1] Nome dado aos testes do RSpec.

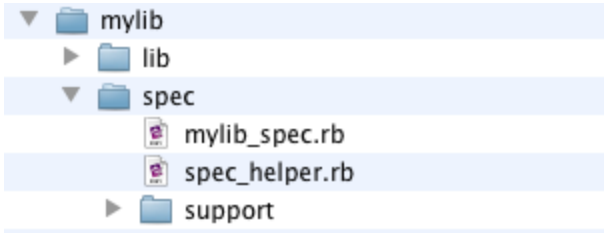

Figura 1: Estrutura convencional do RSpec

O arquivo `spec/spec_helper.rb` é responsável por carregar o RSpec, além de definir dependências e configurações da aplicação e da própria suíte de exemplos. Ele é carregado por todos os arquivos de exemplo.

Seus exemplos são adicionados em arquivos `*_spec.rb`.

O diretório `spec/support` guarda arquivos que você utilizará nos seus exemplos, como novos matchers, exemplos compartilhados, classes, etc.

Um típico arquivo de exemplo se parece com isso:

```
require "spec_helper"

describe MyLib do
end
```

O método `describe` é o responsável por agrupar diversos exemplos e pode conter outros grupos aninhados.

CAPÍTULO 1

Configurando o RSpec

Para configurar o RSpec, você deve utilizar o método `RSpec.configure`, que possui diversas opções, como você pode ver abaixo. Normalmente, estas configurações são adicionadas no arquivo `spec_helper.rb`.

```
RSpec.configure do |config|
  # Ativa output colorido
  config.color_enabled = true

  # Define qual o framework de mocks.
  # Pode ser :rspec, :mocha, :rr, :flexmock, :none
  config.mock_framework = :rspec

  # Ativa o modo de debug.
  # É necessário ter o ruby-debug instalado.
  config.debug = true

  # Exibe backtrace completo caso algum exemplo falhe.
  config.full_backtrace = true

  # Define uma expressão regular que irá evitar que uma
  # linha do backtrace apareça quando um exemplo falha.
  config.backtrace_clean_patterns << /vendor\//

  # Adiciona diretórios ao LOAD_PATH.
  config.libs << "vendor/mylib-1.0.0"

  # Carrega automaticamente os caminhos especificados.
```

```ruby
config.requires << "mylib"

# Inclui um módulo. Os métodos desse módulo estarão disponíveis no
# contexto dos métodos `it`, `specify`, `before`, `after` e `around`.
config.include Module.new

# Inclui um módulo somente nos grupos que casarem o padrão
# especificado pela opção `:file_path`. Esses arquivos irão
# ser definidos como sendo do tipo `:foo`, e irão carregar
# somente módulos deste tipo.
config.include Module.new, :type => :foo, :example_group => {:file_path => /spec\/robots/}

# Estende o grupo de exemplos com o módulo especificado.
# Os métodos deste módulo estão disponíveis no contexto
# dos métodos `describe` e `context`.
config.extend Module.new

# Executa o bloco antes de cada exemplo de um grupo.
config.before {}

# Executa o bloco antes de todos os exemplos de um grupo.
config.before(:all) {}

# Executa o bloco depois de cada exemplo de um grupo.
config.after {}

# Executa o bloco depois de todos os exemplos de um grupo.
config.after(:all) {}

# Executa o bloco antes e depois de cada exemplo de um grupo.
config.around {}
```

```
  # Para a execução da suíte no primeiro exemplo que falhar.
  config.fail_fast = true

  # Executa somente os exemplos que possuirem o filtro `:slow`.
  config.filter_run :slow => true

  # Não executa os exemplos caso o bloco retorne um valor que seja
  # avaliado como `true`.
  config.exclusion_filter = {:ruby => lambda {|version| RUBY_VERSION.to_s !~ /^#{version.to_s}/ }}
end
```

Algumas opções também podem ser configuradas através de um arquivo `.rspec`. Este arquivo deve conter as opções disponíveis no executável `rspec`. Para ver quais são as opções, execute o comando `rspec -h`.

```
$ rspec -h
Usage: rspec [options] [files or directories]

    -I PATH                          Specify PATH to add to $LOAD_PATH (may be used more than once).
    -r, --require PATH               Require a file.
    -O, --options PATH               Specify the path to a custom options file.
        --order TYPE[:SEED]          Run examples by the specified order type.
                                       [default] files are ordered based on the underlying file
                                                 system's order
                                       [rand]    randomize the order of files, groups and examples
                                       [random]  alias for rand
                                       [random:SEED] e.g. --order random:123
        --seed SEED                  Equivalent of --order rand:SEED.
    -d, --debugger                   Enable debugging.
        --fail-fast                  Abort the run on first failure.
        --failure-exit-code CODE     Override the exit code used when there are failing specs.
    -X, --[no-]drb                   Run examples via DRb.
```

```
      --drb-port PORT              Port to connect to the DRb server.
      --init                       Initialize your project with RSpec.
      --configure                  Deprecated. Use --init instead.

**** Output ****

  -f, --format FORMATTER           Choose a formatter.
                                     [p]rogress (default - dots)
                                     [d]ocumentation (group and example names)
                                     [h]tml
                                     [t]extmate
                                     custom formatter class name
  -o, --out FILE                   Write output to a file instead of STDOUT. This option applies
                                     to the previously specified --format, or the default format
                                     if no format is specified.
  -b, --backtrace                  Enable full backtrace.
  -c, --[no-]color, --[no-]colour  Enable color in the output.
  -p, --profile                    Enable profiling of examples and list 10 slowest examples.

**** Filtering/tags ****

  In addition to the following options for selecting specific files, groups,
  or examples, you can select a single example by appending the line number to
  the filename:

    rspec path/to/a_spec.rb:37

  -P, --pattern PATTERN            Load files matching pattern (default: "spec/**/*_spec.rb").
  -e, --example STRING             Run examples whose full nested names include STRING (may be
                                     used more than once)
  -l, --line_number LINE           Specify line number of an example or group (may be
                                     used more than once).
```

```
    -t, --tag TAG[:VALUE]            Run examples with the specified tag, or exclude examples
                                     by adding ~ before the tag.
                                       - e.g. ~slow
                                       - TAG is always converted to a symbol
        --default_path PATH          Set the default path where RSpec looks for examples (can
                                       be a path to a file or a directory).

  **** Utility ****

    -v, --version                    Display the version.
    -h, --help                       You're looking at it.
```

O RSpec irá procurar o arquivo `.rspec` do diretório corrente. Caso ele não encontre, irá procurar na em `~/.rspec`. Veja um exemplo comum de como pode ser este arquivo.

```
--color --fail-fast --format documentation --debug
```

As opções definidas através do método `RSpec.configure` irão sobrescrever as do arquivo `.rspec`. Já o arquivo `.rspec` irá sobrescrever as opções definidas no arquivo `~/.rspec`.

CAPÍTULO 1

Executando exemplos no RSpec

A maneira mais simples de executar os exemplos é através do executável `rspec`. Você só precisa especificar qual o arquivo ou diretório que contém seus exemplos.

```
$ rspec spec/
```

Na versão

Se você não tiver criado nenhum arquivo de exemplo, verá uma mensagem como esta:

```
No examples found.

Finished in 0.00005 seconds
0 examples, 0 failures
```

Você também pode utilizar o Rake. Este é o modo como escolhido pelo RSpec Rails, que ainda define uma tarefa para cada um dos grupos (models, controllers, views, etc).

As opções definidas por padrão irão funcionar muito bem para a maioria dos casos. Basta carregar o arquivo `rspec/core/rake_task.rb` e instanciar a classe. Adicione o código abaixo ao arquivo `Rakefile` (ou equivalente) na raíz do seu projeto.

```
require "rspec/core/rake_task"
RSpec::Core::RakeTask.new
```

Por padrão, o RSpec irá definir a tarefa `spec`, que pode ser executada com o comando `rake spec`.

```
$ rake spec
(in /Users/fnando/rspec-quickguide/code)
No examples matching ./spec/**/*_spec.rb could be found
```

Você pode definir o nome da tarefa.

```
require "rspec/core/rake_task"
RSpec::Core::RakeTask.new(:myspecs)
```

Para ver as tarefas que estão disponíveis, utilize o comando `rake -T`.

```
$ rake -T
(in /Users/fnando/rspec-quickguide/code)
rake myspecs  # Run RSpec code examples
```

A descrição da tarefa pode ser definida com o método `desc` do próprio Rake.

```
require "rspec/core/rake_task"

desc "My custom spec runner"
RSpec::Core::RakeTask.new(:myspecs)
```

Também é possível configurar como o RSpec será executado; basta passar um bloco, como no exemplo abaixo.

```
RSpec::Core::RakeTask.new do |t|
  # Define o nome da tarefa.
  t.name = :spec

  # Define o padrão que será usado para encontrar
```

```ruby
# os arquivos de exemplos. No exemplo abaixo,
# apenas arquivos com a extensão `.spec.rb` serão
# executados.
t.pattern = "spec/**/*/*.spec.rb"

# Define se o Bundler será usado para executar o RSpec.
# O RSpec será executado com o comando `bundle exec`
# caso um arquivo `Gemfile` exista.
t.skip_bundler = false

# Define qual arquivo Gemfile deve ser usado.
t.gemfile = File.expand_path("~/Gemfile")

# Ativa o modo verboso de execução.
t.verbose = true

# Gera o relatório de cobertura de código com RCov.
t.rcov = false

# Define o caminho para o executável do RCov.
t.rcov_path = "/usr/local/bin/rcov"

# Define as opções que o RSpec irá passar para o RCov.
t.rcov_opts = "--exclude 'spec/some_dir'"

# Define o modo como o Ruby será executado. Recebe os mesmos
# argumentos do executável `ruby`.
# No exemplo abaixo, iremos executar o Ruby com a opção `warnings`.
t.ruby_opts = "-w"

# Define as opções do RSpec. Recebe os mesmos argumentos do
# executável `rspec`.
```

```
  t.rspec_opts = ["--color"]
end
```

O RSpec também é integrado com o Autotest. Primeiro, installe a gem autotest-standalone com o comando `gem install autotest-standalone`.

```
$ gem install autotest-standalone
Fetching: autotest-standalone-4.5.9.gem (100%)
Successfully installed autotest-standalone-4.5.9
1 gem installed
```

Depois, crie um arquivo `autotest/discover.rb` na raíz de seu projeto. Este arquivo irá informar ao Autotest que ele deve usar as definições do RSpec para verificar quais arquivos foram modificados. Adicione a linha abaixo:

```
Autotest.add_discovery { "rspec2" }
```

Agora, basta executar o comando `autotest`. Toda vez que um arquivo do diretório `spec` for alterado ou sua contra-parte no diretório `lib`, os exemplos serão executados automaticamente.

```
$ autotest
loading autotest/rspec2
/Users/fnando/.rvm/rubies/ruby-1.9.2-p180/bin/ruby -rrubygems -S
/Users/fnando/.rvm/gems/ruby-1.9.2-p180@global/gems/rspec-core-2.5.1/bin/rspec --tty
'/Users/fnando/Sites/howto-books/rspec-quickguide/code/spec/sample_spec.rb'
.

Finished in 0.00061 seconds
1 example, 0 failures
```

```
# Waiting since 2011-03-23 10:16:16

/Users/fnando/.rvm/rubies/ruby-1.9.2-p180/bin/ruby -rrubygems -S
/Users/fnando/.rvm/gems/ruby-1.9.2-p180@global/gems/rspec-core-2.5.1/bin/rspec --tty
'/Users/fnando/Sites/howto-books/rspec-quickguide/code/spec/sample_spec.rb'
..

Finished in 0.00104 seconds
2 examples, 0 failures

# Waiting since 2011-03-23 10:20:36
```

Para receber notificações toda vez que seus exemplos forem executados, instale a gem Test Notifier.

```
$ gem install test_notifier
Fetching: notifier-0.1.2.gem (100%)
Successfully installed notifier-0.1.2
Successfully installed test_notifier-0.3.6
2 gems installed
```

Carregue o arquivo `test_notifier/runner/rspec.rb` no arquivo `spec/spec_helper.rb`. Ele irá adicionar os hooks necessários para exibir a mensagem.

```
require "test_notifier/runner/rspec"
```

Agora, toda vez que um arquivo for alterado, você irá receber uma notificação como esta:

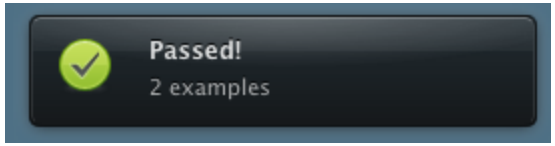

Figura 1: Aviso do Test Notifier no Mac OS X

A gem test_notifier possui suporte para outros meios de notificação para Windows e Linux. Para mais informações, consulte a documentação do projeto.

CAPÍTULO 1

Criando Exemplos

No RSpec, cada teste é chamado de *exemplo*. É neste exemplo que você descreve efetivamente como seu código deve se comportar.

```
describe "Truth" do
  it "should be true" do
    expect(true).to be_true
  end
end
```

O método `describe()` define o *grupo de exemplos*. Dentro de um grupo você pode ter quantos exemplos forem precisos para descrever o comportamento de seu objeto. Estes exemplos são definidos com o método `it()`. A string deste método descreve o comportamento do exemplo.

Na versão 2.11 do RSpec foi adicionado o método `expect`. Este método se tornou o modo recomendado de como seus exemplos devem ser escritos. Antes desta versão, você deveria utilizar os métodos `should` e `should_not`, que eram injetados em todos os objetos do Ruby. Essa abordagem trazia uma série de problemas que foram resolvidos ao isolar o contexto dos exemplos. O mesmo exemplo acima poderia ser escrito da seguinte forma:

```
describe "Truth" do
  it "should be true" do
    true.should be_true
  end
end
```

Conforme anunciado pelos desenvolvedores que mantém o projeto, a sintaxe de `should` e `should_not` não será removida. Mas se você quiser garantir que apenas uma das sintaxes seja utilizada, pode fazer isso através de uma configuração.

```
RSpec.configure do |config|
  config.expect_with :rspec do |config|
    # supports only the new syntax
    config.syntax = :expect

    # supports only the old syntax
    config.syntax = :should

    # supports both syntaxes
    config.syntax = [:should, :expect]
  end
end
```

Embora a sintaxe antiga provalmente jamais deixará de existir, este e-book foi atualizado para a nova versão e irá, sempre que possível, utilizá-la exclusivamente.

Descrevendo objetos e comportamentos

O método `describe` pode receber objetos de qualquer tipo. Adicionalmente, você pode passar um segundo parâmetro que também será usado como parte da descrição.

```
describe("An User") { }
#=> An User

describe(User) { }
#=> User

describe(User, "validations") { }
#=> User validations
```

A grande vantagem de se descrever o comportamento do código de modo correto é que você também cria uma documentação. É possível visualizá-la utilizando o formatter `RSpec::Core::Formatters::DocumentationFormatter`. Para utilizá-lo, execute o RSpec com o argumento `--format documentation`.

```
$ rspec spec/user/validations_spec.rb

User validations
  should validate presence of :name
  should validate presence of :password
  should validate e-mail format of :email
  should validate confirmation of :password

Finished in 0.92756 seconds
4 examples, 0 failures
```

Você pode ter grupos de exemplos aninhados, que podem ser usados para isolar o contexto de seus exemplos.

```
describe User
  describe "validations" do
  end
end
```

Este exemplo irá gerar uma saída como

```
User
  validations
```

O RSpec também possui o método `context()`, que nada mais é que um alias para o método `describe()`.
A única diferença entre eles é semântica; o método `describe()` é mais utilizado para descrever objetos e comportamentos, enquanto o método `context()` é utilizado para descrever o contexto.

O exemplo anterior poderia ser escrito como

```
describe User
  context "when providing valid data"; end
  context "when providing invalid data"; end
end
```

Definindo exemplos na prática

O método `it()` permite especificar um exemplo de código que poderá (ou não) definir as expectativas de um objeto ou comportamento.

Assim como o método `describe()`, o método `it()` aceita uma string que irá descrever as expectativas. Você também pode passar um *hash* que irá definir os filtros, além do bloco. Caso um bloco não seja passado, o exemplo será marcado como pendente.

```
describe "Numbers" do
  it "returns even numbers" do
    numbers = (0..10).select {|n| n % 2 == 0 }
    expect(numbers).to eql([0, 2, 4, 6, 8, 10])
  end

  it "returns odd numbers"
end
```

Ao executar estes exemplos, teremos uma saída como esta:

```
Numbers
  should return even numbers
  should return odd numbers (PENDING: Not Yet Implemented)

Pending:
  Numbers should return odd numbers
    # Not Yet Implemented
    # ./spec/it_spec.rb:9

Finished in 0.00199 seconds
2 examples, 0 failures, 1 pending
```

Existem algumas variações do método `it()`. O RSpec possui os aliases `specify()`[1], que permite descrever uma expectativa quando a construção "it" não faz muito sentido.

```ruby
describe "Numbers" do
  specify "return only odd numbers" do
    numbers = (0..10).select {|n| n % 2 == 1 }
    expect(numbers).to eql([1, 3, 5, 7, 9])
  end
end
```

Além do método `specify()` existem também os métodos `focus()` e `focused()`, que definem o filtro `:focused => true, :focus => true`.

```ruby
describe "Numbers" do
  focus "should return multiples of 5" do
    numbers = (0..10).select {|n| n % 5 == 0 }
```

[1] Os métodos `it()` e `specify()` são apenas alias do método `example()`, embora esse último seja pouco usado.

```
    expect(numbers).to eql([0, 5, 10])
  end

  focused "should return multiples of 7" do
    numbers = (0..21).select {|n| n % 7 == 0}
    expect(numbers).to eql([0, 7, 14, 21])
  end
end
```

Finalmente, o RSpec também adiciona os métodos `xit()` e `pending()`, que definem o filtro `:pending => true` e que não são executados, a menos que você passe explicitamente este filtro na hora que for executar os exemplos.

```
describe "Numbers" do
  xit "should return multiples of 10" do
    (0..100).select {|n| n % 10 == 0}
  end

  pending "should return multiples of 8" do
    (0..24).select {|n| n % 8 == 0}
  end
end
```

Ao executar os exemplos, o RSpec irá exibir uma lista com os que estão pendentes no fim da saída dos resultados.

```
Numbers
  returns multiples of 10 (PENDING: Not Yet Implemented)
  returns multiples of 8 (PENDING: Not Yet Implemented)

Pending:
  Numbers returns multiples of 10
    # Not Yet Implemented
```

```
    # ./spec/it_spec.rb:35
  Numbers returns multiples of 8
    # Not Yet Implemented
    # ./spec/it_spec.rb:39

Finished in 0.00487 seconds
2 examples, 0 failures, 2 pending
```

Exista ainda mais uma maneira de tornar um exemplo pendente; use o método `pending()` dentro de um exemplo, que opcionalmente aceita uma string do motivo da pendência.

```
describe "Numbers" do
  it "should return multiples of 42" do
    pending "The Answer to the Ultimate Question of Life, The Universe, and Everything"
  end
end
```

Hooks: before, after e around

Assim como o Test Unit, o RSpec permite executar alguns *hooks* antes e depois de um exemplo ser executado, mas vai além, como você verá à seguir.

O RSpec possui diversos hooks, que são executados na seguinte ordem:

1. before suite
2. before all
3. before each
4. after each
5. after all
6. after suite

before(:each)

Muitas vezes precisamos preparar o contexto onde iremos executar nossos exemplos. Afinal, você pode querer instanciar algum objeto, definir alguns valores, etc. Para isso existe o hook `before(:each)`.

```
describe "Numbers" do
  before(:each) do
    @numbers = (1..10)
  end
end
```

Por padrão, o método `before()` utiliza o nome `:each`, que pode ser omitido nesse caso.

```
describe "Numbers" do
  before do
    @numbers = (1..10)
  end
end
```

Note que o hook `before(:each)` será executado antes de cada um dos exemplos de um grupo. Por isso, evite realizar tarefas muito demoradas quando a quantidade de exemplos é muito grande. Neste caso, criar um hook com `before(:all)` pode ser uma ideia melhor.

before(:all)

O hook `before(:all)` é executado uma única vez antes de um contexto começar a ser executado. Ele pode ser uma boa alternativa quando você precisa realizar tarefas demoradas, como conectar a um banco de dados. No geral, não é uma boa ideia utilizar o `before(:all)` para compartilhar variáveis com os exemplos[2].

[2] Isso fica claro quando você utiliza o RSpec juntamente com o ActiveRecord. Neste caso, o `before(:all)` não é executado dentro de uma transação, fazendo com que seu banco fique em um estado inconsistente.

```ruby
describe User do
  before(:all) do
    ActiveRecord::Base.establish_connection :adapter => "sqlite3", :database => ":memory:"
  end
end
```

before(:suite)

É executado antes de de qualquer grupo de exemplos ser executado.

after(:each)

O hook `after(:each)` é o complemento do hook `before(:each)` e é executado logo após o término de cada exemplo de um grupo, mesmo quando este exemplo falhar. Por isso, ele pode ser utilizado para restaurar o estado global do ambiente de forma segura.

```ruby
describe "Numbers" do
  after(:each) do
    @numbers = nil
  end
end
```

after(:all)

Existe ainda o hook `after(:all)`, que é executado quando um contexto acabar sua execução. Você pode, por exemplo, encerrar uma conexão com o banco de dados.

```ruby
describe User do
  after(:all) do
    ActiveRecord::Base.remove_connection
  end
end
```

after(:suite)

O hook `after(:suite)` é executado depois que todos os exemplos terminarem sua execução.

around

Existe ainda o caso de quando você quer executar uma ação ANTES e DEPOIS do exemplo. Você poderia criar dois hooks (`before()` e `after()`) com o mesmo bloco. No entanto, existe o hook `around`, que é executado antes e depois de cada exemplo.

```
describe FileUtils do
  around do |example|
    FileUtils.rm_rf("spec/tmp")
    example.run
    FileUtils.rm_rf("spec/tmp")
  end
end
```

O bloco passado para o hook `around()` irá receber o exemplo que deve ser executado com o método `run()`. Mas atenção! Se algum erro acontecer, o código equivalente ao `after()` não será executado. Uma alternativa é executar o hook em um bloco `begin; end`.

```
describe FileUtils do
  around do |example|
    begin
      FileUtils.rm_rf("spec/tmp")
      example.run
    ensure
      FileUtils.rm_rf("spec/tmp")
    end
  end
end
```

Definindo métodos auxiliares (helpers)

Muitas vezes você vai querer utilizar métodos que irão ajudar em algumas tarefas como instanciar um novo objeto, definir algumas variáveis ou preparar o ambiente.

```
describe User do
  it "should set name" do
    user = User.new(:name => "John Doe")
    user.save!

    expect(user.name).to eql("John Doe")
  end

  it "should set e-mail" do
    user = User.new(:email => "john@doe.com")
    user.save!

    expect(user.email).to eql("john@doe.com")
  end
end
```

Ambos os exemplos estão instanciando um novo objeto com algum atributo e executando o método `User#save!` na sequência. Para evitar a duplicação de código desnecessária, você pode definir métodos no escopo do seu contexto. Estes métodos estarão disponíveis nos exemplos e hooks.

```
describe User do
  it "should set name" do
    user = create_user
    expect(user.name).to eql("John Doe")
  end
```

```
  it "should set e-mail" do
    user = create_user
    expect(user.email).to eql("john@doe.com")
  end

  def create_user
    User.new(:name => "John Doe", :email => "john@doe.com").tap do |user|
      user.save!
    end
  end
end
```

Como você pode ver, métodos como este ajudam a reduzir a duplicação de código. No entanto, ele pode se perder em meio aos seus exemplos quando o seu grupo de exemplos for muito grande.
Uma solução é extrair estes métodos para um módulo, e incluí-lo no grupo de exemplos.

```
module UserHelpers
  def create_user
    User.new(:name => "John Doe", :email => "john@doe.com").tap do |user|
      user.save!
    end
  end
end

describe User do
  include UserHelpers

  it "should set name" do
    user = create_user
    expect(user.name).to eql("John Doe")
  end
```

```
  it "should set e-mail" do
    user = create_user
    expect(user.email).to eql("john@doe.com")
  end
end
```

Extrair seus métodos em um módulo à parte tem suas vantagens. Agora, você pode compartilhar tais métodos entre grupos de exemplos diferentes, por exemplo.

Se você quiser tornar este método disponível em todos os grupos de exemplos sem precisar incluir o módulo manualmente, pode utilizar o método `RSpec.configure` para fazer isso globalmente.

```
RSpec.configure do |config|
  config.include UserHelpers
end
```

Exemplos compartilhados

Quando objetos diferentes possuem comportamento semelhante, pode fazer sentido compartilhar exemplos. No RSpec, isso é possível com o método `shared_examples_for`.

```
describe Car do
  describe "Porsche" do
    before { @car = Car.preset(:porsche) }

    it "should have 4 wheels" do
      expect(@car.wheels).to eql(4)
    end
  end
```

```
  describe "Ferrari" do
    before { @car = Car.preset(:ferrari) }

    it "should have 4 wheels" do
      expect(@car.wheels).to eql(4)
    end
  end
end
```

Neste caso, poderíamos compartilhar o exemplo `should have 4 wheels` entre os dois grupos de exemplos.

```
shared_examples_for "car" do
  it "should have 4 wheels" do
    expect(@car.wheels).to eql(4)
  end
end
```

Depois, podemos remover os exemplos duplicados e substituí-los pelo método `it_behaves_like`.

```
describe Car do
  describe "Porsche" do
    before { @car = Car.preset(:porsche) }
    it_behaves_like "car"
  end

  describe "Ferrari" do
    before { @car = Car.preset(:ferrari) }
    it_behaves_like "car"
  end
end
```

Ao executar estes exemplos, você verá que a saída identifica os exemplos como compartilhados.

```
Car
  Porsh
    behaves like car
      should have 4 wheels
  Ferrari
    behaves like car
      should have 4 wheels

Finished in 0.00144 seconds
2 examples, 0 failures
```

Os exemplos compartilhados irão herdar tudo que estiver definido naquele contexto como variáveis de instância, métodos auxiliares, hooks, etc.

CAPÍTULO 1

RSpec::Expectations

O RSpec possui uma nomenclatura diferente do Test Unit quando se refere às *asserções*: *expectativas*. Quando escrevemos exemplos, nós definimos *expectativas* de como nosso código *deve* se comportar. E isso é feito através dos métodos `expect`, `to`, `not_to`/`to_not` e os matchers.

Built-in matchers

O RSpec vem com diversos matchers por padrão, que permitem expressar o comportamento de seu código com uma leitura muito fluente, como você verá abaixo.

be

Passa se o sujeito for avaliado como `true`. Ele também pode ser usado para fazer comparações, utilizando os operadores `>`, `<`, `>=` e `<=`.

```
expect(true).to be
expect(1).to be

expect(1).to be > 0
expect(1).to be >= 0
expect(1).to be < 2
expect(1).to be <= 1

expect(false).not_to be
expect(1).not_to be > 2
```

be_true

Passa se o sujeito for avaliado como `true`.

```
expect(true).to be_true
expect(1).to be_true

expect(false).not_to be_true
expect(nil).not_to be_true
```

be_false

Passa se o sujeito for avaliado como `false`.

```
expect(false).to be_false
expect(nil).to be_false

expect(true).not_to be_false
expect(1).not_to be_false
```

be_nil

Passa se o sujeito for igual a `nil`.

```
expect(nil).to be_nil

expect(false).not_to be_nil
```

be_instance_of
be_an_instance_of

Passa se a classe do sujeito for exatamente a mesma da expectativa.

```
user = User.new

expect(user).to be_instance_of(User)
expect(user).to be_an_instance_of(User)

expect(user).not_to be_instance_of(Integer)
```

be_a
be_an
be_kind_of
be_a_kind_of

Passa se a classe da expectativa for uma superclasse do sujeito.

```
user = User.new

expect(user).to be_an(User)
expect(user).to be_an(Object)

expect(user).not_to be_an(Integer)
```

be_within

Passa se o sujeito for igual à expectativa, considerando a variação do delta.

```
number = 1.56

expect(1.57).to be_within(0.1).of(number)
expect(1.55).to be_within(0.1).of(number)
```

change

Passa se o bloco executado alterar o sujeito de acordo com os parâmetros de comparação (quando definidos).

```
items = []

expect { items << true }.to change(items, :size)
expect { items << true }.to change(items, :size).by(1)
expect { items << true }.to change(items, :size).by_at_least(1)
expect { items << true }.to change(items, :size).by_at_most(1)
expect { items << true }.to change(items, :size).from(4).to(5)

expect { items << true }.to change { items.size }
expect { items << true }.to change { items.size }.by(1)
expect { items << true }.to change { items.size }.by_at_least(1)
expect { items << true }.to change { items.size }.by_at_most(1)
expect { items << true }.to change { items.size }.from(9).to(10)

expect { }.to_not change(items, :size)
expect { }.to_not change { items.size }
```

eq
eql
equal
==
===

Cada um dos métodos de comparação de objetos do Ruby possui uma semântica diferente[1], respeitada pelo RSpec com seus matchers individuais.

[1] Acesse http://www.ruby-doc.org/core/classes/Object.html#M001011 para ver a documentação oficial sobre igualdade no Ruby.

Se você estiver utilizando a sintaxe antiga do RSpec, pode usar o operador `==`, já que queremos apenas comparar se o valor é igual. Ele é semelhante ao método `eql?` na maioria dos casos.

Na dúvida, utilize o matcher `eql` para comparar valores e `equal` para comparar objetos.

```
hello = "hello rspec"

hello.should == "hello rspec"
expect(hello).to eql(hello)
expect(hello).to equal(hello)

expect(hello).to_not equal("hello rspec")

object = Object.new
object.should === object

object.should_not === Object.new
```

exist

Passa se o sujeito responder a um dos métodos `exist?` ou `exists?` e este método retornar um valor avaliado como `true`.

```
expect(Pathname.new(__FILE__)).to exist
expect(File).to exist(__FILE__)

expect(Pathname.new("invalid_file")).not_to exist
```

include

Passa se o sujeito incluir todos os elementos passados na expectativa. Se o sujeito for uma string, irá passar se a substring existir.

```
expect([1, 2, 3]).to include(1)
expect([1, 2, 3]).to include(1,3)

expect(1..3).to include(1)
expect(1..3).to include(1, 2, 3)

expect("hello rspec").to include("lo rs")

expect([1,2,3]).not_to include(4)
expect("hello rspec").not_to include("hi")
```

match

Passa se o sujeito casar a expressão regular que foi passada como expectativa.

```
expect("hello rspec").to match(/hello/)
expect("hello rspec").not_to match(/hi/)
```

=~

O operador `=~` funciona como o matcher `match` quando o sujeito é uma string. Caso o sujeito seja um array, irá passar se todos os itens da expectativa existirem no array, independente da ordem em que estão definidos. Note que este matcher só funciona se você estiver utilizando a syntaxe antiga (`should`/`should_not`).

A construção `should_not` não é implementada quando o sujeito é um array.

```
"hello rspec".should =~ /hello/
"hello rspec".should_not =~ /hi/

[1, 2, 3].should =~ [1, 2, 3]
[1, 2, 3].should =~ [3, 2, 1]
```

raise_error

Passa se o bloco executado lançar uma exceção conforme a expectativa.

```
expect { raise ArgumentError }.to raise_error
expect { raise ArgumentError }.to raise_error(ArgumentError)
expect { raise ArgumentError, "invalid name" }.to raise_error(ArgumentError, "invalid name")
expect { raise ArgumentError, "invalid name" }.to raise_error(ArgumentError, /invalid name/)

expect { }.to_not raise_error
expect { raise StandardError }.to_not raise_error(ArgumentError)
expect { raise ArgumentError, "invalid name" }.to_not raise_error(ArgumentError, "invalid age")
expect { raise ArgumentError, "invalid name" }.to_not raise_error(ArgumentError, /age/)
```

respond_to

Passa se o sujeito responder a todos os nomes passados.

```
expect("hello rspec").to respond_to(:upcase)
expect("hello rspec").to respond_to(:upcase, "downcase")

expect([]).not_to respond_to(:upcase)
```

satisfy

Passa se o retorno do bloco for avaliado como `true`.

```
expect(1).to satisfy {|n| n > 0 }

expect(1).not_to satisfy {|n| n % 2 == 0 }
```

throw_symbol

Passa se o bloco lançar o símbolo especificado na expectativa. Para saber mais como isso funciona, leia a documentação dos métodos catch e throw.

```
expect { throw :halted }.to throw_symbol(:halted)

expect {
  catch(:halted) { throw :halted }
}.to_not throw_symbol
```

Matcher para métodos predicados

Métodos predicados são aqueles terminados com interrogação e que normalmente retornam um valor booleano, como é o caso dos métodos `Array#empty?` e `Fixnum#odd?`, por exemplo.

Uma maneira de você fazer expectativas nestes métodos é utilizando o matcher `be` ou `be_true`, por exemplo.

```
expect(1.odd?).to be

expect([].empty?).to be_true
```

Como você pode perceber, a leitura deste matcher não é tão agradável. Felizmene, o RSpec implementa o matcher `be_<nome do método>`, permitindo que você escreva estes mesmos exemplos de uma forma muito mais clara.

```
expect(1).to be_odd

expect([]).to be_empty
```

Você pode passar argumentos caso o seu método espere algum.

```
expect(File).to be_file(__FILE__)
```

Matcher have

O RSpec também implementa algumas facilidades através do matcher `have`. Todo método começado por `has_<algo>?` pode ser escrito como `have_<algo>`. Veja dois exemplos utilizando os método `Hash#has_key?` e `Hash#has_value?`.

```
expect(:a => 1).to have_key(:a)
expect(:a => 1).to have_value(1)
```

Além disso, também é possível utilizar o matcher `have` para fazer expectativas em coleções de um objeto. Imagine que tenhamos uma classe `Post` que possui uma coleção `Post#comments`.

```
class Post
  attr_accessor :comments

  def initialize
    @comments = []
  end
end
```

Com o matcher `have` podemos expressar diversas expectativas de forma bastante simples.

```
post = Post.new

expect(post).to have(:no).comments

post.comments += ["comment 1", "comment 2"]

expect(post).to have(2).comments
```

```
expect(post).to have_exactly(2).comments
expect(post).to have_at_least(2).comments
expect(post).to have_at_most(2).comments
```

O RSpec também implementa algumas sintaxes puramente cosméticas em arrays e strings, respondendo aos matchers `items()` e `characters()`.

```
expect([]).to have(:no).items
expect([]).to have(0).items
expect([1]).to have(1).item
expect([1,2]).to have(2).items

expect("").to have(:no).characters
expect("hello").to have(5).characters
expect("h").to have(1).character
```

Criando o seu próprio matcher

Como você viu, o RSpec possui muitos matchers. No entanto, às vezes é muito útil criar o seu próprio matcher, fazendo com que seu código fique muito mais expressivo. O RSpec permite criar matchers de duas maneiras, como você verá à seguir.

Usando a DSL

O RSpec permite criar novos matchers muito rapidamente através do método `RSpec::Matchers.define`.

```
it "is multiple of 3" do
  expect(9 % 3).to be_zero
end

it "is multiple of 4" do
```

```
  expect(8 % 4).to be_zero
end
```

Os exemplos acima, embora sejam fáceis de entender, seriam melhores ainda caso eu pudesse simplesmente usar algo como

```
expect(9).to be_multiple_of(3)
expect(8).to be_multiple_of(4)
```

Felizmente, podemos criar nosso próprio matchers com poucas linhas.

```
RSpec::Matchers.define :be_multiple_of do |expected|
  match do |actual|
    actual % expected == 0
  end
end
```

E agora, podemos escrever aqueles mesmos exemplos de forma mais simples.

```
it "is multiple of 3" do
  expect(9).to be_multiple_of(3)
end

it "is multiple of 4" do
  expect(8).to be_multiple_of(4)
end
```

Ao executar estes exemplos, teremos uma saída como esta:

```
Numbers
  is multiple of 3
  is multiple of 4

Finished in 0.00412 seconds
2 examples, 0 failures
```

Perceba que existe ainda um pouco de duplicação em nossos exemplos; estamos usando a mesma descrição no exemplo e no nome do matcher. O RSpec define uma mensagem padrão, já que não especificamos nenhuma em nosso matcher. Mais à frente você como remover esta duplicação.

Existem outros métodos na DSL de matchers. Você definir mensagens personalizadas que serão exibidas quando a expectativa falhar.

```
RSpec::Matchers.define :be_multiple_of do |expected|
  match do |actual|
    actual % expected == 0
  end

  failure_message_for_should do |actual|
    "expected #{actual} to be multiple of #{expected}"
  end

  failure_message_for_should_not do |actual|
    "expected #{actual} not to be multiple of #{expected}"
  end

  description do
    "be multiple of #{expected}"
  end
end
```

Muitos matchers possuem uma *interface fluente*, como é o caso do matcher `be_within`. Você também pode definir uma interface fluente com o método `chain`. Vamos criar um matcher que calcula o tempo de exemplo de um bloco. A API de nosso matcher deve ser algo como

```
expect { some_method }.to take_less_than(3).seconds
```

Além de `seconds`, eu posso utilizar `minutes` como unidade de tempo.

```
RSpec::Matchers.define :take_less_than do |time|
  match do |block|
    started = Time.now
    block.call
    elapsed = Time.now.to_f - started.to_f

    elapsed < (@unit == :minutes ? time.to_f * 60 : time.to_f)
  end

  chain :seconds do
    @unit = :seconds
  end

  chain :minutes do
    @unit = :minutes
  end
end
```

O primeiro passo é criar o matcher principal `take_less_than`. Antes de executarmos o bloco, armazenamos o tempo inicial. E logo após invocarmos o bloco, calculamos a duração da executação. Note que estamos convertendo o método `Time#to_f` pois queremos retornar também os microsegundos.

A DSL de matchers possui o método `chain`, que irá definir a interface fluente. Por baixo dos panos, ele apenas retorna o próprio `self` como resultado do método.

Você também pode definir o método diretamente, mas se esquecer de retornar o `self`, terá resultados inesperados.

```
RSpec::Matchers.define :take_less_than do |time|
  # ...

  def seconds
    @unit = :seconds
    self
  end
end
```

Embora a DSL cubra a maior parte dos casos, você pode querer mais flexibilidade na hora de criar um novo matcher. Para isso, pode criar uma classe que responde a um protocolo pré-definido. O mesmo exemplo poderia ser criado da seguinte forma:

```
module PerformanceMatcher
  class TakeLessThan
    def initialize(time)
      @time = time
    end

    def matches?(block)
      @block = block

      started = Time.now
      block.call
      @elapsed = Time.now.to_f - started.to_f
      @elapsed < (@unit == :minutes ? @time * 60 : @time)
    end
```

```
    def seconds
      @unit = :seconds
      self
    end

    def minutes
      @unit = :minutes
      self
    end

    def failure_message_for_should
      "expected #{@block.inspect} to take less than #{@time} #{@unit}"
    end

    def failure_message_for_should_not
      "expected #{@block.inspect} to take more than #{@time} #{@unit}"
    end
  end

  def take_less_than(time)
    TakeLessThan.new(time)
  end
end

RSpec.configure {|c| c.include(PerformanceMatcher)}
```

Embora seja muito mais verboso (você precisa criar uma classe, torná-la disponível no RSpec, etc), ele permite criar matchers sofisticados de maneira mais organizada.

Note que estamos armazenando os valores em variáveis de instância. Isso é necessário porque, ao contrário da DSL, os valores não são passados para os métodos.

Para definir um novo matcher através de uma classe, você precisa implementar pelo menos o método `matches?` e `failure_message_for_should`. Opcionalmente, você também pode definir outros métodos:

- `does_not_match?`: em alguns casos, é necessário saber se o matcher foi utilizado através do método `should` ou `should_not`. Basta implementar este método que deve retornar `true` para sucesso ou `false` para falha.
- `failure_message_for_should_not`: mensagem utilizada quando você utiliza o `should_not` e o método `matches?` retorna `true`.
- `description`: descrição utilizada quando você utiliza o matcher com o `implicit subject`.

Definindo o sujeito

O RSpec utiliza o objeto que está sendo descrito como o *sujeito* do exemplo, que pode ser acessado através do método `subject()`.

Por padrão, o sujeito é uma instância da classe que está sendo descrita. Se o objeto não for uma classe, então o sujeito será o próprio objeto. O objeto é inicializado em um bloco no hook `before()`.

```
describe User do
  it "is an instance of User" do
    expect(subject).to be_an(User)
  end

  it "is not an admin" do
    expect(subject).not_to be_admin
  end

  it "doesn't set name" do
    expect(subject.name).to be_nil
  end
end
```

No entanto, muitas vezes precisamos definir nosso próprio sujeito, normalmente quando queremos passar argumentos para o método `initialize()`. Neste caso, você pode definir o modo como o sujeito será definido passando um bloco.

```
describe User do
  subject { User.new(:name => "John Doe", :admin => true) }

  it "is an instance of User" do
    expect(subject).to be_an(User)
  end

  it "sets name" do
    expect(subject.name).to eql("John Doe")
  end

  it "is an admin" do
    expect(subject).to be_admin
  end
end
```

Quando utilizamos o método `subject()` corretamente, podemos *delegar* as expectativas para o sujeito, tornando nossos exemplos ainda mais simples.

Delegando para o sujeito

Ao utilizar o sujeito, seja definindo-o explicitamente ou deixando que o RSpec o defina implicitamente, podemos tornar nossos exemplos ainda mais simples.

As expectativas que fizemos na instância da classe `User` podem ficar muito mais simples.

```
describe User do
  context "with default arguments" do
```

```
    it { should be_an(User) }
    it { should_not be_admin }

    its(:name) { should be_nil }
  end

  context "with provided arguments" do
    subject { User.new(:name => "John Doe", :admin => true) }

    it { should be_admin }
    its(:name) { should == "John Doe" }
  end
end
```

Perceba como estamos criando expectativas na instância da classe `User` descrever cada um dos exemplos. E, mesmo sem colocarmos esta informação, o RSpec consegue corretamente definir a descrição à partir do nome do matcher que estamos utilizando.

```
User
  with default arguments
    should be a kind of User
    should not be admin
    name
      should be nil
  with provided arguments
    should be admin
    name
      should == "John Doe"
```

Perceba também que estamos criando expectativas no atributo `User#name` utilizando o método `its()`. Este método permite definir expectativas para qualquer atributo/método do objeto.

let() it be

O RSpec também possui o método `let()`, que permite definir métodos que terão seu valor guardado e da próxima vez que o método for acessado, irá apenas retorná-lo, em vez executar todo o bloco novamente[2].

```
describe User do
  let(:user) { User.new }

  it "is an user" do
    expect(user).to be_an(User)
  end
end
```

[2] Este processo é chamado de Memoize.

CAPÍTULO 1

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 "returns name and email" do
    user = mock(:user, :name => "John Doe", :email => "john@doe.com")

    expect(user.name).to eql("John Doe")
    expect(user.email).to eql("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 "returns name and email" do
    user = mock(:user)
    user.stub :name => "John Doe", :email => "john@doe.com"

    expect(user.name).to eql("John Doe")
    expect(user.email).to eql("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 "responds 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 "returns e-mail" do
    subject.stub :email => "john@doe.com"
    expect(subject.email).to eql("john@doe.com")
  end

  it "returns no e-mail" do
    subject.stub :email
    expect(subject.email).to be_nil
  end

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

    expect(subject.email).to eql("john@doe.com")
  end

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

    expect(subject.email).to eql("john@doe.com")
    expect(subject.name).to eql("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 "returns only one value" do
    subject.stub(:roll_dice).and_return(6)
    expect(subject.roll_dice).to eql(6)
  end

  it "returns different values for each call" do
    subject.stub(:roll_dice).and_return(6,4,1)
    expect(subject.roll_dice).to eql(6)
    expect(subject.roll_dice).to eql(4)
    expect(subject.roll_dice).to eql(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 "returns the recent things count" do
    subject.stub_chain(:things, :recent, :count => 10)
    expect(subject.things.recent.count).to eql(10)
  end

  it "returns the older things count" do
    subject.stub_chain(:things, :older, :count).and_return(15)
    expect(subject.things.older.count).to eql(15)
```

```
  end
end
```

any_instance

O RSpec 2.6.0 introduziu um novo método chamado `any_instance()`. Este método permite fazer stubbing ou criar expectativas em qualquer instância criada à partir de uma determinada classe.

```
describe "User class" do
  it "saves record" do
    User.any_instance.stub(:save).and_return(true)

    user = User.new
    expect(user.save).to be

    another_user = User.new
    expect(user.save).to be
  end
end
```

Note que você precisa definir o stubbing utilizando a construção `stub(:method).and_return(value)`. Caso contrário, seu exemplo irá falhar[1].

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`.

[1] Particularmente acho que isto é um bug.

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

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

Executando blocos

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

```
describe User do
  it "yields" do
    User.stub(:create).and_yield(subject)

    User.create do |user|
      expect(user).to eql(subject)
    end
  end
end
```

Message expectation

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

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

    expect(subject.email).to eql("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 "doesn't 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 "instantiates 1 user" do
    User.should_receive(:new).once
    User.new
  end

  it "instantiates 3 users" do
    User.should_receive(:new).exactly(3).times
```

```
    3.times { User.new }
  end

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

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

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

  it "instantiates at least 1 user" do
    User.should_receive(:new).at_least(1).times
    3.times { User.new }
  end

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

  it "doesn't 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 "initializes 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 "initializes 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 "initializes 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 "initializes 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 "initializes 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 "sets 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 "sets 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 "sets 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 "is not an 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 "calls 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 "initializes 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 "initializes 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 para a opção `RSpec.configuration.mock_framework`. Neste caso, ele deve definir 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.

CAPÍTULO 1

Usando RSpec com Ruby on Rails

O RSpec possui suporte nativo para o Ruby on Rails através da extensão rspec-rails. Esta extensão permite criar expectativas em diversas partes do Ruby on Rails como modelos, controllers, views e helpers, tudo isso de forma isolada.

Configurando o ambiente

O Rails 3 permite configurar uma aplicação com RSpec de forma bastante simples. À partir da raíz de seu projeto, edite o arquivo `Gemfile`. Adicione um novo grupo com o seguinte conteúdo, caso ele ainda não exista. Seu arquivo deve se parecer com isto:

```
source :rubygems

gem "rails", "3.2.8"
gem "sqlite3"

group :development, :test do
  gem "rspec-rails"
  gem "capybara"
end
```

Note que precisamos adicionar o RSpec ao grupo `:development` para que as rake tasks possam ser executadas sem precisarmos passar a variável de ambiente `RAILS_ENV=test` a todo momento.

Para instalar as bibliotecas, execute o comando `bundle install`.

Agora, podemos executar o gerador do RSpec, que irá copiar alguns arquivos e diretórios necessários para que o RSpec funcione.

```
$ rails generate rspec:install
    create  .rspec
    create  spec
    create  spec/spec_helper.rb
```

Se este é um projeto novo e você ainda não escreveu nenhum teste usando o Test Unit, remova o diretório `RAILS_ROOT/test`, já que o RSpec utiliza o diretório `RAILS_ROOT/spec`.

Para ver se tudo está funcionando, execute os comandos abaixo.

```
$ rake db:migrate
$ rake spec
```

Você deverá ver algo como isto:

```
$ rake spec
(in /myapp)
No examples matching ./spec/**/*_spec.rb could be found
```

Modelos

Modelos possuem um comportamento muito próximo de bibliotecas Ruby convencionais, onde você pode executar diretamente os métodos. Por exemplo, para escrever expectativas quanto à validação de um modelo, podemos utilizar o matcher `have(n).errors_on`.

Vamos criar o modelo `Link`, que irá armazenar endereços de um serviço de encurtamento de urls.

À partir da raíz de seu projeto, execute o comando `rails generate model Link url:text`. Depois, execute o comando `rake db:migrate`.

```
$ rails generate model Link url:text
      invoke  active_record
      create    db/migrate/20110409110422_create_links.rb
      create    app/models/link.rb
      invoke    rspec
      create      spec/models/link_spec.rb

$ rake db:migrate
(in /Users/fnando/short_url)
==  CreateLinks: migrating ====================================================
-- create_table(:links)
   -> 0.0015s
==  CreateLinks: migrated (0.0016s) ===========================================
```

Perceba que o RSpec gera automaticamente os arquivos de teste, como é o caso de `spec/models/link_spec.rb`. Abra este arquivo e adicione expectativas de como nosso modelo deverá se comportar.

```
require "spec_helper"

describe Link do
  context "validations" do
    it "requires url to be set"

    it "is valid with valid url"

    it "is not valid when url is invalid"
  end
end
```

Execute o comando `rake spec` ou `rake spec:models`.

```
$ rake spec
(in /Users/fnando/short_url)
/Users/fnando/.rvm/rubies/ruby-1.9.2-p180/bin/ruby -S bundle exec rspec
***

Pending:
  Link validations requires url to be set
    # Not Yet Implemented
    # ./spec/models/link_spec.rb:6
  Link validations is valid with valid url
    # Not Yet Implemented
    # ./spec/models/link_spec.rb:8
  Link validations is not valid when url is invalid
    # Not Yet Implemented
    # ./spec/models/link_spec.rb:10

Finished in 0.00044 seconds
3 examples, 0 failures, 3 pending
```

Agora podemos implementar nossa primeira expectativa.

```
it "requires url to be set" do
  link = Link.create
  expect(link).to have(1).error_on(:url)
end
```

Ao executar os testes mais um vez, teremos algo como `3 examples, 1 failure, 2 pending`.
Nossa expectativa exige que o atributo `#url` seja definido. Isso pode ser garantido se adicionarmos uma validação no modelo.

```
class Link < ActiveRecord::Base
  validates_presence_of :url
end
```

Execute os testes novamente. Você terá algo como `3 examples, 0 failures, 2 pending`. Agora, passe para o próximo exemplo pendente.

```
it "is valid with valid url" do
  link = Link.create(:url => "http://example.org")
  expect(link).to have(:no).errors_on(:url)
end
```

Ao executar os testes teremos algo como `3 examples, 0 failures, 1 pending`. Por último, podemos escrever a expectativa que diz que o modelo deve ser inválido se passarmos uma url inválida.

```
it "is not valid when url is invalid" do
  link = Link.create(:url => "invalid")
  expect(link).to have(1).error_on(:url)
end
```

Execute os testes; o resultado deverá ser algo como `3 examples, 1 failure`. Em nosso modelo, vamos fazer uma verificação simples apenas para saber se existe o *schema* `http://` ou `https://`. Adicione uma nova validação, agora com a macro `validates_format_of`.

```
class Link < ActiveRecord::Base
  validates_presence_of :url
  validates_format_of :url, :with => %r[\Ahttps?://]i, :allow_blank => true
end
```

Execute os testes pela última vez. Você deverá receber algo como `3 examples, 0 failures`.

Controllers

Agora que nosso modelo já está funcionando como deveria, podemos partir para o controller, que irá agir como um intermediador entre o modelo e a view.

O RSpec também gera os arquivos relacionados ao controller quando você executa o gerador do Rails. Execute o comando `rails generate controller links new show`.

```
$ rails generate controller links new show
      create  app/controllers/links_controller.rb
       route  get "links/show"
       route  get "links/new"
      invoke  erb
      create    app/views/links
      create    app/views/links/new.html.erb
      create    app/views/links/show.html.erb
      invoke  rspec
      create    spec/controllers/links_controller_spec.rb
      create    spec/views/links
      create    spec/views/links/new.html.erb_spec.rb
      create    spec/views/links/show.html.erb_spec.rb
      invoke  helper
      create    app/helpers/links_helper.rb
      invoke    rspec
      create      spec/helpers/links_helper_spec.rb
```

Toda vez que você gera um controller, ele irá gerar o arquivo de expectativas para o controller, helper e views (caso sejam definidas). Dessa forma, você evita desperdiçar tempo criando os arquivos manualmente.

Vamos criar nossas expectativas quanto ao comportamento do controller. Abra o arquivo `spec/controllers/links_controller_spec.rb` e adicione os exemplos abaixo.

```
require "spec_helper"

describe LinksController do
  describe "POST 'create'" do
    it "creates link"

    it "saves link"

    it "redirects to link overview"

    it "displays message"
  end
end
```

Execute os exemplos acima com o comando `rake spec` ou `rake spec:controllers`. Você irá receber algo como `4 examples, 0 failures, 4 pending`. Vamos implementar nosso primeiro exemplo.

```
it "creates link" do
  Link.should_receive(:new).with("url" => "http://example.org")
  post :create, :link => {:url => "http://example.org"}
end
```

Execute o comando `rake spec:controller`.

```
$ rake spec:controllers

Pending:
  LinksController POST 'create' redirects to link overview
    # Not Yet Implemented
    # ./spec/controllers/links_controller_spec.rb:12
```

```
Failures:

  1) LinksController POST 'create' creates link
     Failure/Error: post :create, :link => {:url => "http://example.org"}
     ActionController::RoutingError:
       No route matches {:link=>{:url=>"http://example.org"}, :controller=>"links", :action=>"create"}
     # ./spec/controllers/links_controller_spec.rb:8:in `block (3 levels) in <top (required)>'

Finished in 0.06976 seconds
2 examples, 1 failure, 1 pending
```

Nosso exemplo falhou dizendo que não existe nenhuma rota para a ação que foi executada. Abra o arquivo `config/routes.rb` e adicione algo como

```
ShortUrl::Application.routes.draw do
  resources :links
end
```

Execute os testes mais uma vez. Desta vez nosso exemplo irá falhar pois ainda não criamos a ação `create`. Abra o arquivo `app/controllers/links_controller.rb` e adicione o método `create`.

```
class LinksController < ApplicationController
  def create
  end
end
```

Ao rodar os testes novamente, você irá perceber que ele falhou porque não criamos nosso arquivo de view, `app/views/links/create.html.erb`, e nem iremos criar! Isso porque toda vez que um link for criado, o usuário será redirecionado para a página daquele link. Vamos tornar exemplo pendente temporariamente, passando para o próximo exemplo.

```
  it "creates link" do
    pending "depends on redirection"

    Link.should_receive(:new).with(:url => "http://example.org")
    post :create, :link => {:url => "http://example.org"}
  end
```

Nosso segundo exemplo especifica que devemos ser redirecionado para a página com detalhes sobre o link. Vamos implementá-lo.

```
  it "redirects to link overview" do
    link = mock_model(Link)
    Link.stub(:new).and_return(link)

    post :create
    expect(response).to redirect_to(:action => "show", :id => link.id)
  end
```

Neste exemplo, estamos criando um mock do modelo `Link` com o método `mock_model`. Este método irá retornar um objeto mock baseado no modelo, com diversos métodos definidos pelo ActiveRecord, tornando possível criar expectativas mesmo quando modelos ainda não foram completamente implementados.

Nosso controller pode ser implementado da seguinte maneira:

```
  class LinksController < ApplicationController
    def create
      link = Link.new(params[:link])
      redirect_to link_path(link)
    end
  end
```

Ao executar os testes, você deverá receber algo como `4 examples, 0 failures, 3 pending`.

Agora que já implementamos o redirecionamento, podemos voltar ao primeiro exemplo. Remova a linha `pending "depends on redirection"` e execute os testes. Você irá receber uma mensagem dizendo que a rota `show` não foi encontrada. Isso acontece porque estamos definindo uma expectativa no método `Message.new`, mas não estamos retornando nenhum objeto e o Rails não consegue inferir qual é a rota sem um `id`. Vamos fazer como no exemplo `redirects to link overview`, retornando um objeto mock.

```
it "creates link" do
  link = mock_model(Link)
  Link.should_receive(:new).with("url" => "http://example.org").and_return(link)

  post :create, :link => {:url => "http://example.org"}
end
```

Execute os testes mais uma vez. Agora, você deve receber algo como `4 examples, 0 failures, 2 pending`.

Perceba que existe um ponto de duplicação este os dois exemplos que escrevemos até agora; ambos definem um objeto mock. Agora é uma boa hora para utilizar o método `let()`. Seu exempo ficará parecido com

```
describe Link do
  let(:link) { mock_model(Link) }

  describe "POST 'create'" do
    it "creates link" do
      Link.should_receive(:new).with("url" => "http://example.org").and_return(link)
      post :create, :link => {:url => "http://example.org"}
    end

    it "redirects to link overview" do
      Link.stub(:new).and_return(link)
```

```
      post :create
      expect(response).to redirect_to(:action => "show", :id => link.id)
    end
  end
end
```

Agora, podemos implementar o exemplos que diz que o link deve ser salvo. Nós iremos utilizar a mesma abordagem do exemplo `create link` para definir uma expectativa com o método `should_receive`.

```
it "saves link" do
  Link.stub(:new).and_return(link)
  link.should_receive(:save).and_return(true)

  post :create
end
```

Execute os testes. Você deverá receber algo como `4 examples, 1 failure, 1 pending`. Nosso exemplo está falhando porque ainda não estamos executando o método `Link#save`. No controller, basta fazer esta implementação.

```
class LinksController < ApplicationController
  def create
    link = Link.new(params[:link])
    link.save
    redirect_to link_path(link)
  end
end
```

Execute os testes. Você vai perceber que um teste que estava passando inicialmente começou a falhar.

```
1) LinksController POST 'create' creates link
     Failure/Error: post :create
       Mock "Link_1001" received unexpected message :save with (no args)
     # ./app/controllers/links_controller.rb:4:in `create'
     # ./spec/controllers/links_controller_spec.rb:10:in `block (3 levels) in <top (required)>'
```

Isso acontece porque estamos chamando um método em um objeto mock que não espera tal chamada. Você pode evitar este tipo de erro com o método `as_null_object`. Altere a definição do mock para `let(:link) { mock_model(Link).as_null_object }`.

Execute os testes mais uma vez. O resultado deverá ser algo como `4 examples, 0 failures, 1 pending`.

Agora, apenas o último exemplo ainda não foi escrito. Toda vez que um novo link for criado, devemos exibir uma mensagem.

```
it "displays message" do
  Link.stub(:new).and_return(link)
  link.stub(:save => true)
  post :create

  expect(flash[:notice]).to eql("Your link has been shortened!")
end
```

Ao executar os testes, veremos que ele falhou.

```
1) LinksController POST 'create' displays message
     Failure/Error: flash[:notice].should == "Your link has been shortened!"
       expected: "Your link has been shortened!"
            got: nil (using ==)
     # ./spec/controllers/links_controller_spec.rb:34:in `block (3 levels) in <top (required)>'
```

Nossa implementação é bastante trivial; basta adicionar uma chamada ao `flash[:notice]` em nossa ação.

```
class LinksController < ApplicationController
  def create
    link = Link.new(params[:link])
    link.save
    flash[:notice] = "Your link has been shortened!"
    redirect_to link_path(link)
  end
end
```

Execute os testes uma última vez e você verá todos os exemplos são executados com sucesso!

Podemos reduzir um ponto de duplicação de código nestes exemplos. Estamos utilizando o método `Message.stub` em mais de um lugar; podemos mover esta definição para um bloco `before()`. Veja como ficou nossos exemplos após esta refatoração:

```
require "spec_helper"

describe LinksController do
  let(:link) { mock_model(Link).as_null_object }

  describe "POST 'create'" do
    before do
      Link.stub(:new).and_return(link)
      link.stub(:save => true)
    end

    it "creates link" do
      Link.should_receive(:new).with("url" => "http://example.org").and_return(link)
      post :create, :link => {:url => "http://example.org"}
    end

    it "redirects to link overview" do
      post :create
```

```
      expect(response).to redirect_to(:action => "show", :id => link.id)
    end

    it "saves link" do
      link.should_receive(:save)
      post :create
    end

    it "displays message" do
      post :create
      expect(flash[:notice]).to eql("Your link has been shortened!")
    end
  end
end
```

Da maneira como implementamos nosso controller, estamos assumindo que todos os links são válidos, já que não estamos tratando o caso de falhas. Mova todos os exemplos que criamos até agora para dentro de um contexto:

```
require "spec_helper"

describe LinksController do
  let(:link) { mock_model(Link).as_null_object }

  describe "POST 'create'" do
    before do
      Link.stub(:new).and_return(link)
    end

    it "creates link" do
      Link.should_receive(:new).with("url" => "http://example.org").and_return(link)
      post :create, :link => {:url => "http://example.org"}
```

```
    end

    it "saves link" do
      link.should_receive(:save)
      post :create
    end

    context "when link is saved" do
      before do
        link.stub(:save => true)
      end

      it "redirects to link overview" do
        post :create
        expect(response).to redirect_to(:action => "show", :id => link.id)
      end

      it "displays message" do
        post :create
        expect(flash[:notice]).to eql("Your link has been shortened!")
      end
    end
  end
end
```

Crie um novo contexto que irá conter todos os casos de falha. Ele será parecido com isto:

```
  context "when link is not saved" do
    before do
      link.stub(:save => false)
    end
```

```
    it "assigns @link"

    it "renders the new template"

    it "does not display message"
  end
```

Execute os testes. Você deverá receber algo como `7 examples, 0 failures, 3 pending`. Vamos implementar nosso primeiro exemplo.

```
  it "assigns @link" do
    post :create
    expect(assigns[:link]).to eql(link)
  end
```

Este exemplo verifica se uma variável de instância `@link` foi definida. Ela será usada para popular o formulário em caso de erros. Execute os testes e você verá que ele irá falhar:

```
  1) LinksController POST 'create' when link is not saved assigns @link
       Failure/Error: expect(assigns[:link]).to eql(link)

         expected #<Link:0x80d5c764 @name="Link_1005">
              got nil

         (compared using eql)
```

Vá ao controller e altere todas as referências à variável `link` para `@link`.

```
class LinksController < ApplicationController
  def create
    @link = Link.new(params[:link])
    @link.save
    flash[:notice] = "Your link has been shortened!"
    redirect_to link_path(@link)
  end
end
```

Ao executar os testes você deverá ver algo como `7 examples, 0 failures, 2 pending`.

Nosso segundo exemplo irá verificar se estamos vendo a página com o formulário, em vez de sermos redirecionados.

```
it "renders the new template" do
  post :create
  expect(response).to render_template("new")
end
```

Execute os testes e você verá que ele falhou. Nossa implementação exige uma condição para saber se deve-se redirecionar ou renderizar a página.

```
class LinksController < ApplicationController
  def create
    @link = Link.new(params[:link])

    if @link.save
      flash[:notice] = "Your link has been shortened!"
      redirect_to link_path(@link)
    else
      render :new
    end
```

```
  end
end
```

Ao executar os testes, teremos apenas 1 exemplo pendente. Este exemplo irá verificar se a mensagem é exibida ou não.

```
it "does not display message" do
  post :create
  expect(flash[:notice]).to be_nil
end
```

Execute os testes mais uma vez. Você verá que todos eles passaram e nosso controller está devidamente testado. Se você estiver utilizando o formato `documentation`, verá que a saída da suíte servirá muito bem como uma especificação de como nossa aplicação deve se comportar.

```
LinksController
  POST 'create'
    creates link
    saves link
    when link is saved
      redirects to link overview
      displays message
    when link is not saved
      assigns @link
      renders the new template
      does not display message

Finished in 0.20989 seconds
7 examples, 0 failures
```

Veja como ficaram nossos exemplos finais:

```
require "spec_helper"

describe LinksController do
  let(:link) { mock_model(Link).as_null_object }

  describe "POST 'create'" do
    before do
      Link.stub(:new).and_return(link)
    end

    it "creates link" do
      Link.should_receive(:new).with("url" => "http://example.org").and_return(link)
      post :create, :link => {:url => "http://example.org"}
    end

    it "saves link" do
      link.should_receive(:save)
      post :create
    end

    context "when link is saved" do
      before do
        link.stub(:save => true)
      end

      it "redirects to link overview" do
        post :create
        expect(response).to redirect_to(:action => "show", :id => link.id)
      end

      it "displays message" do
        post :create
```

```
        expect(flash[:notice]).to eql("Your link has been shortened!")
      end
    end

    context "when link is not saved" do
      before do
        link.stub(:save => false)
      end

      it "assigns @link" do
        post :create
        expect(assigns[:link]).to eql(link)
      end

      it "renders the new template" do
        post :create
        expect(response).to render_template("new")
      end

      it "does not display message" do
        post :create
        expect(flash[:notice]).to be_nil
      end
    end
  end
end
```

Views

Agora que nosso controller já está criando links, podemos criar expectativas em nossas views, garantindo que a informação seja exibida corretamente. Como views são alvo de mudanças ao longo do tempo, é muito importante que você foque estes testes na informação exibida, e não na estrutura HTML.

Nosso exemplo irá garantir que tanto o link original quanto a versão encurtada seja, exibidos em nossa página.

```
describe "links/show.html.erb" do
  it "displays original link" do
    expect(rendered).to contain("http://google.com")
  end

  it "displays shortened link" do
    expect(rendered).to contain("http://test.host/16i")
  end
end
```

A versão encurtada da url é verificada com o domínio `http://test.host`; este domínio é definido pelo próprio RSpec e identifica que você utilizou um método `*_url` em vez de `*_path`.

Execute os testes com o comando `rake spec:views` para vê-los falhar! O método `rendered()` irá retornar o conteúdo da view, renderizado pelo método `render()`. Adicione esta chamada a um bloco `before()`, já que ambos os exemplos irão utilizá-lo.

Execute os testes novamente e você verá que eles continuarão falhando. Isso acontece porque nossa view possui o conteúdo gerado pelo próprio Rails. Abra o arquivo `app/views/links/show.html.erb` e altere seu conteúdo para algo como isto:

```
<h1>short.ly</h1>
<p>
  The shortened version of <%= @link.url %> is
```

```
  <%= expand_link_url @link.id.to_s(32) %>.
</p>
```

Embora nossa view tenha bastante lógica que poderia ser extraída para outros lugares, ela servirá muito bem ao nosso propósito. Depois, podemos refatorá-la. Execute os testes e você verá que agora iremos receber uma mensagem de erro.

```
1) links/show.html.erb displays original link
     Failure/Error: render
     ActionView::Template::Error:
       undefined method `url' for nil:NilClass
     # ./app/views/links/show.html.erb:3:in `_app_views_links_show_html_erb__1976788296286281169_216912[illegible]
     # ./spec/views/links/show.html.erb_spec.rb:5:in `block (2 levels) in <top (required)>'
```

Isso acontece porque nossa view utiliza uma variável de instância `@link`, que é uma instância da classe `Link`. Nós podemos definir variáveis de instância com o método `assign()`. Nós iremos definir esta variável com um objeto mock que retorna apenas os métodos usados em nossa view.

```
before do
  assign :link, mock(:link, :url => "http://google.com", :id => 1234)
  render
end
```

Execute os testes. O motivo da falha agora é o método `expand_link_path`, que utiliza uma rota ainda não definida. Edite o arquivo `config/routes.rb`, adicionando esta nova rota.

```
ShortUrl::Application.routes.draw do
  resources :links
  get ":shortlink", :to => "links#expand", :as => :expand_link
end
```

Execute os testes e você verá que eles irão passar.

Helpers

Nosso exemplo define a versão encurtada da url na própria view. Podemos retirar um pouco desta lógica da view e passá-la para um helper. Primeiro, vamos criar o nosso exemplo validando nossa intenção. Abra o arquivo `spec/helpers/links_helper_spec.rb` e adicione nosso exemplo.

```
describe LinksHelper do
  it "returns shortened url" do
    link = mock(:link, :id => 1234)
    expect(helper.shortened_url(link)).to eql("http://test.host/16i")
  end
end
```

Execute os testes com o comando `rake spec:helpers` e você verá que este exemplo irá falhar.

```
1) LinksHelper returns shortened url
     Failure/Error: expect(helper.shortened_url(link)).to eql("http://test.host/i")
     NoMethodError:
       undefined method `shortened_url' for #<#<Class:0x00000100e34d00>:0x00000100db44e8>
     # ./spec/helpers/links_helper_spec.rb:6:in `block (2 levels) in <top (required)>'
```

Agora, abra o arquivo `app/helpers/links_helper.rb` e adicione o nosso método `shortened_url`, extraindo a lógica da view para este método.

```
module LinksHelper
  def shortened_url(link)
    expand_link_url(link.id.to_s(32))
```

```
  end
end
```

Execute os testes mais uma vez e você verá que este exemplo não irá mais falhar. Agora, precisamos alterar o arquivo `app/views/links/show.html.erb`, substituindo aquele lógica por esta chamada ao método `shortened_url`.

```
<h1>short.ly</h1>
<p>
  The shortened version of <%= @link.url %> is
  <%= shortened_url @link %>.
</p>
```

Execute os testes das views com o comando `rake spec:views`. Você verá que o comportamento não foi alterado e seus testes continuarão passando!

Requests

Embora tenhamos testado cada um dos componentes de nossa aplicação isoladamente, você deve garantir que seu sistema está funcionando como um todo. Isso pode ser feito com testes de integração, que irão agir em um nível mais alto de abstração, semelhante ao modo como o usuário veria sua aplicação.

O RSpec 2 introduziu um novo grupo de exemplos chamado `requests`. Este grupo de exemplos verifica se você tem o Webrat ou o Capybara instalado e utiliza como driver dos testes de integração.

Vamos criar um novo exemplo que irá executar o fluxo completo desde a criação até a visualização do link que criamos:

1. acessamos a página de criação
2. preenchemos o formulário e clicamos no botão de envio
3. somos redirecionados para a tela de visualização

O RSpec não possui geradores para criar os testes de integração. Crie o arquivo `spec/requests/create_link_spec.rb`. Adicione a estrutura inicial, como o exemplo abaixo.

```
require "spec_helper"

describe "Create link" do
end
```

Nosso exemplo irá utilizar o Capybara, já configurado na primeira parte deste capítulo. Inicie um novo bloco `before()`. Este bloco terá todas as ações que serão executadas até o momento onde iremos fazer nossas expectativas.

```
before do
  visit "/links/new"

  fill_in "Url", :with => "http://google.com"
  click_button "Shorten"
end
```

Primeiro, nós visitamos a página que contém o formulário (`/links/new`). Depois, preenchemos o formulário e clicamos no botão de submit.

Agora, vamos adicionar os exemplos que irão validar nosso comportamento.

```
it "displays long url"

it "displays shortened url"
```

Execute os testes com o comando `rake spec:requests` e você verá algo como `2 examples, 0 failures, 2 pending`. Vamos implementar nosso primeiro exemplo.

```
  it "displays long url" do
    expect(page).to have_content("http://google.com")
  end
```

Este exemplo irá verificar se o link original está sendo exibido. Toda página que for acessada está disponível no método `response.body`. Execute os testes e você que ele falhou. Isso acontece porque nós ainda não implementamos a view `app/links/new.html.erb`, embora tenhamos criado exemplos que a utilizavam. Abra este arquivo e adicione algo como isto:

```
  [missing 'code/short_url/app/views/links/new.html.erb' file]
```

Execute os testes novamente. Eles ainda estão falhando. Desta vez, o motivo é que ainda não implementamos a ação `LinksController#new`, que deve instanciar um novo objeto da classe `Link`. Abra o arquivo `app/controllers/links_controller.rb` e adicione o método `new()`.

```
  def new
    @link = Link.new
  end
```

Agora, execute os testes mais uma vez. Ele falhou novamente pois também não implementamos a ação `LinksController#show`. Adicione o método abaixo ao arquivo `app/controllers/links_controller.rb`.

```
  def show
    @link = Link.find(params[:id])
  end
```

Ao executar os testes, você verá algo como `2 examples, 0 failures, 1 pending`. Agora, vamos passar ao próximo exemplo, que irá verificar se o link encurtado está sendo exibido.

```
  it "displays shortened url" do
    expect(page.body).to match(%r[http://www\.example.com/[a-zA-Z0-9]+])
  end
```

Ao contrário das views e helpers, o domínio utilizado pelo grupo `requests` é `http://www.example.com`. Execute os testes e você verá que ele também passará.

CAPÍTULO 1

Outras bibliotecas

Fakeweb

O FakeWeb permite forjar requisições HTTP no Ruby. Ela age diretamente sobre a biblioteca Net, sem que você precise modificar o seu código ou criar stubs.

Você pode instalá-la com o comando `gem install fakeweb`. No seu arquivo `spec/spec_helper.rb`, carregue o FakeWeb, adicionando a linha `require "fakeweb"`.

Como é recomendado que você não faça requisições HTTP, você pode desabilitar todo e qualquer acesso a recursos externos. Para isso, utilize o método `FakeWeb.allow_net_connect`, definindo seu valor como `false`. Se você tentar fazer uma requisição que não foi registrada, receberá uma exceção `FakeWeb::NetConnectNotAllowedError`.

Para registrar uma requisição, você deve utilizar o método `FakeWeb.register_uri`.

```
it "parses response" do
  FakeWeb.register_uri(:get, "http://simplesideias.com.br", :body => "Hello")
  open("http://simplesideias.com.br").read.should contain("Hello")
end
```

Após registrar esta requisição, sempre que você acessar o endereço `http://simplesideias.com.br` com o método `GET`, o retorno será a mensagem `Hello`.

Óbviamente este exemplo não mostra um uso real. O que você fazer, em vez de retornar uma string, é salvar o conteúdo da URL que está registrando em um arquivo. Depois, você pode ler este arquivo e utilizar seu conteúdo como retorno da requisição.

```
it "parses response" do
  body = File.read("spec/fixtures/simplesideias.html")
  FakeWeb.register_uri(:get, "http://simplesideias.com.br", :body => body)
  open("http://simplesideias.com.br").read.should contain("Simples Ideias")
end
```

FakeFS

Assim como requisições HTTP, operações que envolvem o disco rígido pode ser muito lentas. Além disso, quando você começa a fazer muito stubbing em métodos de uma biblioteca, seu teste fica muito acoplado à implementação, em vez de garantir que o *comportamento* é que está correto.

Foi pensando nisso que o Chris Wanstrath, fundador do Github criou o FakeFS, uma biblioteca que forja chamadas que envolvam arquivos e diretórios através das bibliotecas `File`, `Dir` e `FileUtils` do Ruby.

Para instalá-la, execute o comando `gem install fakefs`. Adicione-a ao seu arquivo `spec/spec_helper.rb` com a linha `require "fakefs"`.

Agora, toda e qualquer chamada que envolva operações com arquivos e diretórios serão forjadas. Isso significa que o exemplo abaixo não irá criar um diretório de fato, mas mesmo assim ainda é possível criar expectativas.

```
it "creates directory" do
  FileUtils.mkdir_p("/tmp/some/path")
  File.should be_directory("/tmp/some/path")
end
```

Se você não quiser, não precisa substituir o comportamento padrão por completo. Neste caso, basta ativar o FakeFS apenas em um determinado contexto, por exemplo. Altere seu arquivo `spec/spec_helper.rb` para carregar a biblioteca usando `require "fakefs/safe"`.
Agora pode usar os métodos `FakeFS.activate!` e `FakeFS.deactivate!`.

```
it "creates directory" do
  FakeFS.activate!
  FileUtils.mkdir_p("/tmp/some/path")
  File.should be_directory("/tmp/some/path")
  FakeFS.deactivate!
  FakeFS::FileSystem.clear
end
```

Perceba que estamos utilizando o método `FakeFS::FileSystem.clear`; assim, limpamos os estado do FakeFS, evitando inconsistências.

Delorean

Uma outra coisa que exige stubbing a todo momento são operações que envolvam hora, como o método `Time.now`. Uma biblioteca que permite fazer isso de modo menos manual é a Delorean.

Para instalá-la, execute o comando `gem install delorean`. Depois, carregue a biblioteca em seu arquivo `spec/spec_helper.rb` e inclua os helpers para RSpec.

```
require "delorean"

RSpec.configure do |config|
  config.include Delorean
end
```

Agora você pode utilizar o método helper `time_travel_to`.

```
it "returns newest users first" do
  time_travel_to("1 month ago") { User.create(:name => "Mary") }
```

```
    time_travel_to("3 months ago") { User.create(:name => "John") }

    User.newest.first.name.should == "Mary"
  end
```

Factory Girl

O RSpec possui suporte a fixtures de banco de dados quando usado com Rails. O grande problema das fixtures é quando você precisa criar objetos que são associados a outros, por exemplo. Além disso, as fixtures são carregadas diretamente no banco de dados, não passando por validações ou callbacks, por exemplo.

Foi pensando nisso que a Thoughtbot lançou o Factory Girl, uma biblioteca que substitui as fixtures com diversas funcionalidades como herança de factories, criação de atributos sequenciais e associações.

Para instalá-la, execute o comando `gem install factory_girl`. Depois, carregue a biblioteca no arquivo `spec/spec_helper.rb` com a linha `require "factory_girl"`. Depois, você já pode definir suas factories com o método `Factory.define`.

Crie um arquivo em `spec/support/factories.rb`. O primeiro argumento do método `Factory.define` deve ser um símbolo que identifica sua factory. Por padrão ele considera este argumento como sendo o modelo que ele deverá utilizar. Sendo assim, a definição `factory(:user) {}` irá criar factories para o modelo `User`.

```
  Factory.define do
    factory :user do
      name "John Doe"
      email "john@doe.com"
    end
  end
```

Para criar o objeto no banco de dados, você deve utilizar o método `FactoryGirl.create()`, passando o nome da factory que vai utilizar.

```
john = FactoryGirl.create(:user)
```

Você pode sobrescrever os atributos na hora que for instânciar a factory. Basta passar um hash.

```
mary = FactoryGirl.create(:mary, :name => "Mary")
```

A factory `:user` define sempre os mesmos atributos. Se você criar dois usuários com a configuração padrão, ambos irão utilizar o e-mail "john@doe.com". Se você tiver alguma validação de unicidade, sua factory irá lançar uma exceção na hora que for instanciada.

```
class User < ActiveRecord::Base
  validates_uniqueness_of :email
end

user1 = FactoryGirl.create(:user)
user2 = FactoryGirl.create(:user)
#=> Lançará a exceção ActiveRecord::RecordInvalid
```

Nesse caso, você pode utilizar o método `sequence` para criar e-mails únicos.

```
FactoryGirl.define do
  factory :user do
    sequence(:email) {|i| "john#{i}@doe.com" }
  end
end
```

Perceba que estamos utilizando um bloco para definir o próximo item da sequência. Se você não fizer isso, ele irá retornar um e-mail para todos, como nossa implementação anterior. Você também deve utilizar um bloco se precisar definir datas, por exemplo.

O Factory Girl também suporta associações.

```
FactoryGirl.define
  factory :project do
    name "My Project"
    association :user
  end
end
```

Se você instanciar a factory `:project`, o Factory Girl irá definir automaticamente uma associação com uma factory `:user`, a menos que você defina o objeto que deve ser usado.

```
project1 = FactoryGirl.create(:project)
#=> cria um novo objeto User

user = FactoryGirl.create(:user)
project2 = FactoryGirl.create(:project, :user => user)
#=> usa um objeto User existente
```

Atenção: o uso de factories pode deixar sua suíte muito lenta. Principalmente quando você tem objetos com muitas associações definidas, mesmo que você não esteja utilizando. Idealmente, seria bom ter algo com a facilidade do Factory Girl com a velocidade das fixtures. Acontece que existe, como você verá à seguir!

Factory Girl Preload

Para resolver este problema de lentidão quando sua suíte faz uso intensivo de factories, criei uma biblioteca chamada Factory Girl Preload. A ideia é utilizar o Factory Girl para dar carga no banco de dados antes de executar a suíte de testes. Desse modo, você pode reaproveitar uma factory entre os exemplos.

Para instalá-la, execute o comando `gem install factory_girl-preload`. Carregue a biblioteca adicionando a linha `require "factory_girl/preload"` ao arquivo `spec/spec_helper.rb`.

Depois de definir suas factories, você pode definir quais factories são pré-criadas em banco de dados. Para isso, utilize o método `preload`.

```
FactoryGirl.define do
  preload do
    factory(:john) { Factory(:user) }
    factory(:myapp) { Factory(:project, :user => users(:john)) }
  end
end
```

Simples assim! Essas factories serão criadas antes da suíte ser executada. Agora, em seus exemplos, você pode pode simplesmente referenciar a factory que foi pré-carregada.

```
it "has permission on a given project" do
  projects(:myapp).should be_accessible_by(users(:john))
end
```

Note que cada modelo terá seu próprio método de acesso às factories. Esse método é inferido à partir do nome do modelo. Sendo assim, o modelo `User` terá um método `users()`, o modelo `Project` terá um método `projects()`, e assim por diante.

SimpleCov

O SimpleCov é uma ferramenta que analiza a cobertura de código no Ruby 1.9, identificando trechos que não foram testados, utilizando a própria biblioteca do Ruby para isso.

Para instalá-la, utilize o comando abaixo:

```
$ gem install simplecov
```

No seu arquivo `spec/spec_helper.rb`, carregue o SimpleCov. Para isso, adicione as linhas abaixo antes de carregar o seu próprio código.

```
require "simplecov"
SimpleCov.start
```

Se você quer utilizar o SimpleCov com o Ruby on Rails, lembre-se de iniciá-lo passando o nome do adaptador.

```
require "simplecov"
SimpleCov.start "rails"
```

Agora, toda vez que você executar sua suíte de exemplos, um diretório `coverage` será regerado. Para visualizar sua cobertura de código, abra o arquivo `coverage/index.html`.

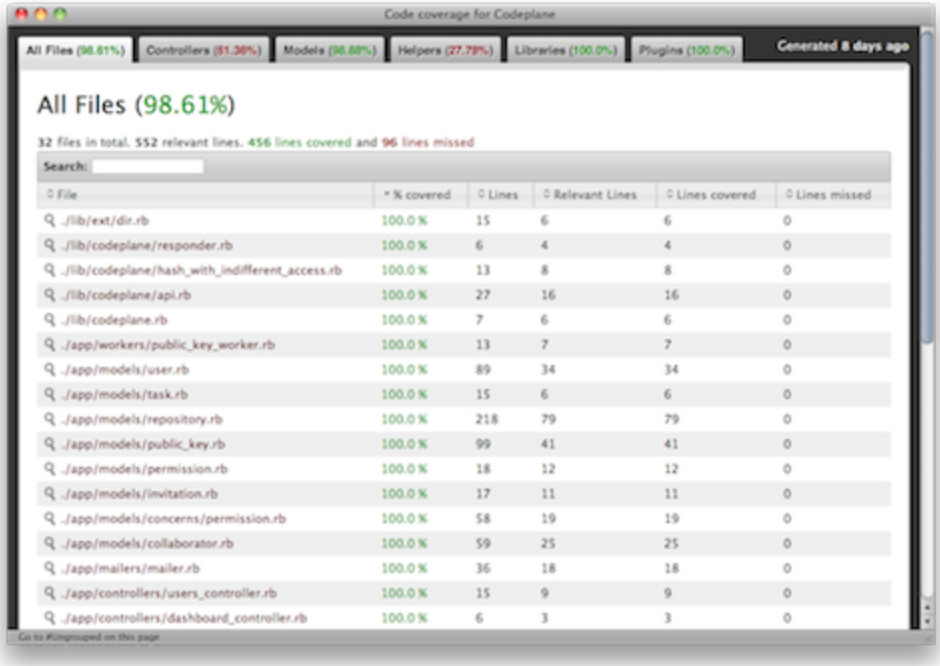

Figura 1: Resultado de cobertura do SimpleCov

Se você *ainda* utiliza o Ruby 1.8, dê uma olhada no RCov, que faz algo semelhante.

Cucumber

O Cucumber é uma ferramenta que executa testes de aceitação em texto puro ou, melhor dizendo, em linguagem natural. O Cucumber permite que você escreva histórias em inglês ou português, por exemplo. Isso permite que clientes, que não possuem conhecimento técnico, possam descrever funcionalidades que podem ser implementadas pelo time de desenvolvimento. Na prática, isso acontece muito pouco.

Para instalar o Cucumber, execute o comando abaixo.

```
$ gem install cucumber
```

O Cucumber pode ser usado em diversos contextos. Embora seja mais comum utilizá-lo para criar testes de aceitação para aplicativos Ruby/Rails, ele pode ser usado com outras linguagens como, por exemplo, o PHP.

Você verá como escrever uma história utilizando o Cucumber com Ruby. Primeiro, crie um novo diretório chamado `calculator`. Crie o arquivo `calculator/Rakefile` com as linhas abaixo; elas irão gerar uma rake task que executará nossas *features*.

```
require "cucumber/rake/task"

Cucumber::Rake::Task.new do |t|
  t.cucumber_opts = %w[--format pretty --color]
end
```

Agora, crie o diretório `calculator/features`; é nele que iremos colocar todos os arquivos reconhecidos pelo Cucumber. Nossa primeira funcionalidade será a adição. Crie um arquivo `calculator/features/adicao.feature`.

```
# language: pt
Funcionalidade: Adição
  Para fazer cálculos matemáticos
  Como um usuário
  Eu quero saber a soma de dois números

  Esquema do Cenário: Somar dois números
    Dado que eu informei o número <numero_1>
    E que eu informei o número <numero_2>
    Quando eu pressiono o botão <botao>
    Então o resultado deve ser <resultado>
```

```
  Exemplos:
    | numero_1  | numero_2  | botao | resultado  |
    | 1         | 1         | sum   | 2          |
    | 5         | 15        | sum   | 20         |
    | 29        | 1         | sum   | 30         |
```

Para definir qual o idioma utilizado em sua funcionalidade, basta adicionar o comentário `# language: pt`. Se quiser saber quais são as palavras-chave utilizadas em um determinado idioma, por exemplo o português, execute o comando `cucumber --i18n pt`.

```
$ cucumber --i18n pt
      | feature          | "Funcionalidade"                          |
      | background       | "Contexto"                                |
      | scenario         | "Cenário", "Cenario"                      |
      | scenario_outline | "Esquema do Cenário", "Esquema do Cenario" |
      | examples         | "Exemplos"                                |
      | given            | "* ", "Dado "                             |
      | when             | "* ", "Quando "                           |
      | then             | "* ", "Então ", "Entao "                  |
      | and              | "* ", "E "                                |
      | but              | "* ", "Mas "                              |
      | given (code)     | "Dado"                                    |
      | when (code)      | "Quando"                                  |
      | then (code)      | "Então", "Entao"                          |
      | and (code)       | "E"                                       |
      | but (code)       | "Mas"                                     |
```

O Cucumber utiliza um formato mais ou menos padronizado para definir uma funcionalidade. Normalmente, você terá algo como:

```
Funcionalidade: [alguma funcionalidade]
  Para [fazer algo]
  Como [alguém]
  Eu quero [alguma coisa]

  Cenário: [alguma situação]
    Dado [alguma ação]
    Quando [alguma ação]
    Então [resultado das ações anteriores]
```

Execute as funcionalidades com o comando `rake cucumber`. O Cucumber irá avisar que você não criou alguns `step definitions`, que definem como irão funcionar os passos `Given-When-Then` e, inclusive, irá sugerir alguns snippets de código que podem ser utilizados.

Antes de implementarmos o step definition, vamos preparar o ambiente, carregando a classe `Calculator` e o módulo de expectativas do RSpec. Crie o arquivo `calculator/features/support/env.rb`. Esse arquivo é carregado automaticamente pelo Cucumber e deve preparar o ambiente.

```
require "rspec/expectations"
require "calculator"
```

Agora, crie a classe `Calculator` no arquivo `calculator/lib/calculator.rb`. O diretório `lib` é adicionado automaticamente ao `$LOAD_PATH`.

```
class Calculator
end
```

Finalmente, crie um arquivo `calculator/features/step_definitions/calculator_steps.rb`. Ele será responsável por definir como nossa funcionalidade será definida. Primeiro, vamos instanciar uma nova calculadora, que será usada em nossos steps. Isso pode ser feito com o método `Before()`.

```
Before do
  @calc = Calculator.new
end
```

Agora, podemos criar nossos step definitions. Você pode utilizar o formato sugerido, em português se quiser. Particularmente, prefiro manter todo o meu código em inglês, abrindo exceção para as features do Cucumber.

Adicione o step definition para a expressão `Dado que informei o número`. O Cucumber detecta *givens* subsequentes quando utilizando o conector `E/And`; por isso, só precisamos definir o step apenas uma vez.

```
Given /que eu informei o número (\d+)/ do |number|
  @calc << number.to_i
end
```

Defina um novo atributo `numbers`, que é um array vazio por padrão.

```
class Calculator
  attr_accessor :numbers

  def initialize
    @numbers = []
  end
end
```

Depois, adicione o método `Calculator#<<`, que irá adicionar os números que serão utilizados no cálculo.

```
class Calculator
  # ...

  def <<(number)
```

```
    numbers << number
  end
end
```

Adicione o step definition para a expressão `Quando eu pressionar o botão`, que deve executar o método de mesmo nome. Assim, quando pressiono o botão `sum`, devo executar o método `Calculator#sum`.

```
When /eu pressiono o botão (.+)/ do |button|
  @result = @calc.send(button)
end
```

Perceba que estamos armazenando o resultado do método `Calculator#sum` na variável de instância `@result`; nós iremos utilizar essa variável no próximo step definition. Implemente o método `Calculator#sum`, que irá somar todos os números.

```
class Calculator
  # ...

  def sum
    numbers.inject(:+)
  end
end
```

Adicione agora o step definition da expressão `Então o resultado deve ser`, que irá efetivamente fazer a expectativa quanto ao resultado esperado.

```
Then /o resultado deve ser (\d+)/ do |result|
  @result.should == result.to_i
end
```

Ao executar o comando `rake cucumber`, você verá que sua funcionalidade passou!

```
$ rake cucumber
Funcionalidade: Adição
  Para fazer cálculos matemáticos
  Como um usuário
  Eu quero saber a soma de dois números

  Esquema do Cenário: Somar dois números     # features/adicao.feature:7
    Dado que eu informei o número <numero_1> # features/step_definitions/calculator_steps.rb:10
    E que eu informei o número <numero_2>    # features/step_definitions/calculator_steps.rb:10
    Quando eu pressiono o botão <botao>      # features/step_definitions/calculator_steps.rb:16
    Então o resultado deve ser <resultado>   # features/step_definitions/calculator_steps.rb:22

    Exemplos:
      | numero_1 | numero_2 | botao | resultado |
      | 1        | 1        | sum   | 2         |
      | 5        | 15       | sum   | 20        |
      | 29       | 1        | sum   | 30        |

3 scenarios (3 passed)
12 steps (12 passed)
0m0.009s
```

O Cucumber também é integrado ao Ruby on Rails. Para ter mais informações sobre o Cucumber e como utilizá-lo com o Rails, veja o wiki do projeto no Github.