RSpec

Crie especificações executáveis em Ruby

Casa do Código

HE:labs

MAURO GEORGE

Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Agradecimentos

Muito obrigado à minha Mãe e ao meu Pai, Maria Nilza e José Mauro, por me apoiarem em tudo e estarem sempre ao meu lado.

Como o livro não poderia existir sem o Ruby, obrigado ao Matz e ao Ruby Core team por criar uma linguagem orientada a objetos tão divertida e expressiva. E ao David Heinemeier Hansson e todo o Rails team por criar um framework web que torna o nosso trabalho muito mais divertido. E a todos os criadores, mantenedores e contribuidores das ferramentas que veremos aqui como WebMock, VCR e factory_girl.

Obrigado a todos da Casa do Código por acreditarem no projeto e ajudarem a torná-lo realidade. Especialmente ao Adriano e à Vivian, por terem que lidar diretamente com o meu excelente conhecimento de português.

Obrigado à HE:labs por tudo o que tenho aprendido e produzido. Obrigado ao Rafael Lima e ao Sylvestre Mergulhão, donos da HE:labs, por apoiarem o meu projeto do livro e torná-lo uma parceria com a HE:labs.

Obrigado ao Cayo Medeiros(Yogodoshi), por sempre me dar excelentes dicas em qualquer novo projeto que inicio ou iniciamos juntos.

Obrigado à Jéssica, minha namorada e futura esposa. =) Por me aturar há mais de 6 anos e sempre estar ao meu lado me apoiando nas minhas decisões e fazendo a minha vida muito mais feliz!

Sobre o Autor

Mauro George atualmente é desenvolvedor de software na HE:labs, onde trabalha com Ruby e Agile no seu dia a dia. Com mais de seis anos de experiência com desenvolvimento web, tem contribuído em diversos projetos open source, incluindo o shoulda-matchers, já tendo feito uma pequena contribuição no Rails. É palestrante e apaixonado por vídeo game desde sempre, também e sócio do Estou Jogando.

A HE:labs

A HE:labs faz produtos web fantásticos desde a ideia ao lançamento. Isso inclui aplicativos, e-commerce, sistemas corporativos, aplicativos em cloud e qualquer tipo de software para internet. Utilizando de Design Thinking, Lean Startup e Desenvolvimento Ágil para transformar a sua ideia em um produto real. Possui uma equipe multidisciplinar composta por programadores, analistas, designers e hackers. Para conhecer um pouco mais, acesse: http://helabs.com.br.

Introdução

Seja bem-vindo! O Ruby tem um excelente ambiente de testes com ferramentas como MiniTest, RSpec e Cucumber. Mas o que podemos melhorar no nosso TDD do dia a dia? Qual o próximo passo? Além de padrões e convenções bem definidas que podemos seguir, temos diversas ferramentas que ajudam enquanto escrevemos nossos testes. Trabalhando em diversos times e em projetos open source, pude notar que certos testes eram escritos de forma complicada, sem padrões, apenas *happy paths* eram testados, projetos com baixa cobertura de testes, falta de conhecimento de bibliotecas que ajudariam no processo de teste etc.

E destes e outros problemas surgiu a ideia do livro: mostrar técnicas e ferramentas que tendem a melhorar o nosso processo de escrever testes.

Sendo assim o objetivo do livro é que você saia daqui:

- escrevendo melhor os seus testes, seguindo convenções e boas práticas que serão abordadas;
- esteja equipado com uma coleção de ferramentas e técnicas para quando enfrentar problemas durante o processo de TDD.

Eu vou aprender a escrever testes em Ruby?

O livro não é uma introdução ao RSpec ou TDD. Assume-se que o leitor tenha conhecimento do RSpec ou qualquer outra biblioteca de testes em Ruby e que esteja habituado ao processo de TDD.

Utilizo o Minitest/Test::Unit, este livro serve para mim?

Sim. Utilizamos o RSpec nos exemplos, mas muitas das técnicas e ferramentas podem ser utilizadas independente da ferramenta de testes. Lembrando que também passamos por características exclusivas do RSpec.

Sobre a abordagem

Utilizamos na maior parte do tempo TDD enquanto estamos escrevendo os exemplos, começando pelo teste e seguindo para a implementação.

Propositalmente, utilizamos um domínio da aplicação simples, dado que o nosso foco aqui é na parte dos testes.

Utilizamos uma aplicação em Ruby on Rails, dado que a maioria dos iniciantes começa pelo Rails para só depois aprender Ruby. Sendo assim, os mais experientes não terão dificuldade de acompanhar os exemplos mesmo que nunca tenham trabalhado com Rails, bem como os iniciantes que estão acostumados apenas com o ambiente do Rails.

Os exemplos de código podem ser encontrados em https://github.com/maurogeorge/rspecbf-exemplos.

Todo o código é escrito em português, devido ao fato de o livro ser em português. No entanto, recomendo que se você não escreve o seu código em inglês procure fazê-lo no próximo projeto, afinal os projetos open source possuem seu código e documentação escritos originalmente em inglês. Além do mais, é bem provável que você venha a trabalhar com alguém de outro país e que esta pessoa tenha uma grande chance de não falar o nosso português.

Sumário

1 O bom e velho RSpec **1**

1.1 Bah, mas por que testar? 1
1.2 Meu primeiro teste, agora com RSpec 5
1.3 O tal do RSpec 6
1.4 A sintaxe de expectativa 6
1.5 Descrevendo bem o seu teste 10
1.6 Não teste apenas o *happy path* 11
1.7 Definindo o sujeito 12
1.8 No dia a dia não se esqueça de 15
1.9 Conclusão 17

2 Testes que acessam rede... WTF!?! **19**

2.1 Introdução 19
2.2 Consumindo uma API 20
2.3 WebMock ao resgate 24
2.4 Utilizando o cURL 26
2.5 Mas eu quero automatizar isso... 28
2.6 VCR??? É o videocassete de que meu pai fala? 29
2.7 Utilizando o VCR 29
2.8 Dados sensíveis no VCR 35
2.9 URIs não determinísticas 38
2.10 Conclusão 43

3 Fixtures são tão chatas! Conheça a factory_girl 45
3.1 Introdução . . . 45
3.2 Instalação . . . 46
3.3 Criando nossa primeira factory . . . 47
3.4 Utilizando a factory . . . 47
3.5 Factories nos testes . . . 48
3.6 Sendo DRY . . . 51
3.7 Atributos dinâmicos nas factories . . . 55
3.8 Associações . . . 58
3.9 Bah, mas só funciona com Active Record? . . . 65
3.10 Conhecendo as estratégias . . . 67
3.11 E quando as factories não são mais válidas? . . . 70
3.12 Conclusão . . . 72

4 Precisamos ir... de volta para o futuro 73
4.1 Introdução . . . 73
4.2 Congelando o tempo com timecop . . . 75
4.3 Removendo repetição . . . 78
4.4 Rails 4.1 e o ActiveSupport::Testing::TimeHelpers . . . 81
4.5 Conclusão . . . 82

5 Será que testei tudo? 83
5.1 Introdução . . . 83
5.2 O falso 100% . . . 85
5.3 Meu objetivo é ter 100% de cobertura de testes? . . . 89
5.4 Mas e você senhor autor, quanto faz de cobertura de testes? . . . 91
5.5 Conclusão . . . 92

6 Copiar e colar não é uma opção! 95
6.1 Introdução . . . 95
6.2 O shared example . . . 96
6.3 Criando um Matcher . . . 99
6.4 O shoulda-matchers . . . 103
6.5 Matchers de terceiros . . . 108
6.6 Conclusão . . . 109

7 O tal dos mocks e stubs **111**
7.1 Conhecendo o stub 111
7.2 Os dublês 114
7.3 Expectativas em mensagens 119
7.4 Matchers de argumentos 124
7.5 Um pouco mais sobre stubs, dublês e message expectations . 126
7.6 Mockar ou não mockar? 129
7.7 Conclusão 129

8 Não debugamos com puts, certo? **131**
8.1 Por um melhor console 132
8.2 Conhecendo o Pry 133
8.3 Conclusão 138

9 Conclusão **139**

Bibliografia **141**

Capítulo 1

O bom e velho RSpec

1.1 Bah, mas por que testar?

Se você ainda não sabe quais as vantagens de ter testes automatizados no seu sistema, vamos a uma historinha.

Mauro e Rodrigo trabalham em um projeto que possui um cadastro de produto que é feito em três passos. Hoje, o cadastro de produto funciona muito bem, no entanto é preciso adicionar um novo passo entre o segundo e o terceiro. Mauro e Rodrigo começam a criar este novo passo, mas o processo é bastante repetitivo e passivo a erro, dado que depois que o usuário atinge um passo não pode voltar. Sendo assim, a cada linha alterada e a cada tentativa de implementar algo, eles devem passar por todo o processo para só depois conseguir testar. Além do feedback lento, a cada vez que alteram um passo recebem um erro inesperado na tela devido à falta de parâmetro de um método ou um objeto que não foi instanciado e gera `NoMethodError`:

`undefined method 'metodo'for nil:NilClass` para todo o lado.

Se nossos heróis continuarem assim, não obterão sucesso nesta jornada. É uma história bem comum de um time de pessoas que não utiliza testes unitários. Vamos ver como estes testes podem nos ajudar:

- **Feedback constante**: a cada vez que rodamos nossos testes, sabemos se uma funcionalidade está funcionando ou não em segundos.
- **Fácil manutenção**: caso algo esteja errado, o teste quebrará, sendo assim o desenvolvedor tem total confiança de alterar algo.
- **Reduz bugs**: como testamos cada um dos nossos métodos unitariamente, temos casos de pontas garantindo que cada uma das partes do software se comporte como devido.
- **Melhor design**: assim como temos confiança de adicionar coisa nova, também temos total confiança de refatorar o que já foi feito.
- **Documentação**: cada teste é uma especificação de como um método ou uma parte do sistema funciona, ficando disponível e atualizado para todos do time.
- **Divertido**: além de todas as vantagens, escrever testes é sempre divertido.

A comunidade Ruby é muito focada em testes, projetos open source como o `rails`, `devise`, `cancan` etc., que possuem seus testes automatizados. Na realidade a exceção são projetos que não possuem testes.

Então, sem desculpas: vamos escrever nosso primeiro teste.

Meu primeiro teste

O Ruby já possui na sua biblioteca padrão seu framework de testes chamado de `Test::Unit`, no entanto, a partir do Ruby 2.1.0 é recomendado o uso do MiniTest para quando estamos criando um projeto novo. O Minitest também já faz parte da biblioteca padrão do Ruby.

Em aplicações rails temos definido o `ActiveSupport::TestCase`, que é a classe de testes que devemos herdar para definirmos nossos testes.

O `ActiveSupport::TestCase` herda de `MiniTest::Unit::TestCase`, sendo assim, quando estamos testando uma app rails sem definir qual o nosso framework de testes, por padrão estamos utilizando o Minitest. Em versões mais antigas era usado o `Test::Unit`.

Chega para nós a seguinte missão: Vamos criar um app que simula uma batalha entre pokémons. No entanto, primeiro temos que preencher nossa base de dados com os pokémons. Para isso criamos o nosso model `Pokemon`.

Vamos escrever um teste para o model `Pokemon` que possui um método `#nome_completo`. Ele simplesmente retorna o valor do `nome` e do `id_nacional` que cada pokémon possui definido, mas primeiro vamos ao teste.

Iniciamos nosso teste primeiro criando um arquivo em `test/models/pokemon_test.rb`, no qual incluímos o arquivo `test_helper.rb`. Em seguida, criamos uma classe com o nome do nosso model acrescido de `Test`, o `PokemonTest` que herda de `ActiveSupport::TestCase`. Com o mínimo definido, vamos agora criar o teste para o nosso método. Para isso, definimos um método com o prefixo `test` que indica ao minitest que aquele é um teste. No método prefixado por `test` criamos uma instância de pokémon e utilizamos do método `assert_equal` para realizarmos o teste. Como primeiro parâmetro, passamos o valor que esperamos como retorno e no segundo, o método que estamos testando.

```
require 'test_helper'

class PokemonTest < ActiveSupport::TestCase

  def test_exibe_o_nome_e_o_id_nacional
    pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
    assert_equal 'Charizard - 6', pokemon.nome_completo
  end
end
```

Definido o nosso teste, vamos executá-lo com `$ rake test test/models/pokemon_test.rb`. Como ainda não implementa-

mos este método, nosso teste irá falhar. Vamos agora implementá-lo para fazer o nosso teste passar.

Para isso criamos o método `Pokemon#nome_completo` e interpolamos os valores `nome` e `id_nacional`.

```
class Pokemon < ActiveRecord::Base

  def nome_completo
    "#{nome} - #{id_nacional}"
  end
end
```

Ao executarmos o teste novamente, ele passa. O próximo passo seria refatorarmos o nosso método, mas como ele faz pouca coisa, não temos o que refatorar. Este é o chamado TDD (Desenvolvimento guiado a testes), que consiste em seguirmos os passos de primeiro escrever o teste, depois escrever o mínimo de código para o teste passar e, por último, refatorarmos.

O rails define o método `test`, que nos permite passarmos uma string e um bloco e, automaticamente, é gerado o método com o prefixo *test* como fizemos anteriormente. Para alteramos o nosso teste anterior para utilizar o método `test`, removemos a definição de método, utilizamos o `test` e passamos a ele uma string em vez de um nome separado por _(underline) e um bloco com o nosso teste.

```
test 'exibe o nome e o id nacional' do
  pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
  assert_equal 'Charizard - 6', pokemon.nome_completo
end
```

É melhor utilizar o método `test` pois ele nos gera mais legibilidade.

Legal! Mas por que o RSpec?

Devido ao fato de o RSpec incentivar testes mais legíveis por conta de sua sintaxe. Na comunidade Ruby temos os que preferem testar utilizando o Minitest / `Test::Unit` [1] e os que preferem o RSpec [7]. Independente do framework que escolher, o mais importante é que estamos testando. Vamos em frente e veremos como podemos fazer o mesmo teste utilizando o RSpec.

1.2 Meu primeiro teste, agora com RSpec

Instalando

Adicionamos ao nosso Gemfile.

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

Em seguida, rodamos `$ bundle install`. É importante que a gem esteja nos grupos `development` e `test`.

Finalizado o bundle, rodamos `$ rails generate rspec:install` para gerar o diretório `spec/` e gerar os arquivos de configuração do RSpec.

O teste

Vamos escrever o mesmo teste que fizemos para o `Pokemon#nome_completo`, no entanto, agora utilizando o RSpec. Criamos o nosso arquivo de teste em `spec/models/pokemon_spec.rb`, no qual incluímos o `spec_helper.rb`. Iniciamos dizendo qual a classe que estamos testando, passando para o método `describe` a nossa classe `Pokemon`, além de um bloco, que é onde definiremos todos os nossos testes. Definimos o nosso primeiro teste utilizando o método `it`, que recebe uma string com o nome do nosso teste e um bloco, que é onde ele é realizado. Para fazermos a asserção do nosso teste, utilizamos o método `should` que o RSpec cria em nosso objetos, e em seguida passamos para o *matcher* `eq` ,que define que nossa asserção deve ser igual ao valor passado ao `eq`.

```
it 'exibe o nome e o id nacional' do
  pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
  pokemon.nome_completo.should eq('Charizard - 6')
end
```

Até o momento, ele não difere muito do Minitest, ainda mais se utilizarmos o método `test` que o rails nos fornece. Mas não se preocupe: veremos ainda neste capítulo algumas dicas de como melhorar a legibilidade de nossos testes utilizando todo o poder do RSpec.

1.3 O tal do RSpec

Com o primeiro release em 18 de maio de 2007, o RSpec tem bastante história e diversos colaboradores. Quase atingindo a sua versão 3.0 estável, muita coisa aconteceu como mudança de sintaxe e o surgimento de um padrão de boas práticas criado pela comunidade. Sendo assim, muitos exemplos de códigos em posts mais antigos podem estar usando a sintaxe antiga, além de não seguir as boas práticas. Vamos dar uma olhada nessas práticas para assim utilizarmos sempre o melhor que o RSpec tem a nos oferecer.

1.4 A sintaxe de expectativa

Por muito tempo, o RSpec utilizou do que chamamos de sintaxe `should`, como acabamos de ver.

```
it 'exibe o nome e o id nacional' do
  pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
  pokemon.nome_completo.should eq('Charizard - 6')
end
```

A partir do RSpec 2.11 foi incluída a sintaxe *expect*. Para utilizá-la, simplesmente passamos o nosso objeto para o método `expect` e, em seguida, utilizamos o método `to`. Vamos alterar o nosso teste para usá-la.

```
it 'exibe o nome e o id nacional' do
  pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
  expect(pokemon.nome_completo).to eq('Charizard - 6')
end
```

É mais verboso que a clássica sintaxe `should`, que tentava ser bem próxima a uma frase em inglês.

Um dos principais motivos desta mudança de sintaxe foi devido à sua implementação que utilizava *monkey patch* no `Kernel`. Se sua classe herda de outra que não inclui o módulo `Kernel`, como `BasicObject`, você receberá um erro `NoMethodError` se estiver utilizando a sintaxe `should`.

Monkey Patch

No Ruby, as classes são abertas, o que na prática quer dizer que podemos mudar o comportamento de uma classe a qualquer momento durante a execução do programa. Podemos alterar, adicionar e remover métodos. Vamos a um exemplo.

No Ruby não temos um método que simplesmente soma todos os valores do array e retorna um inteiro. Podemos fazer isso utilizando o `inject`, então vamos criar o nosso método `Array#soma` que soma todos os valores de um array.

Para isso simplesmente definimos um método na nossa classe `Array` e nele utilizamos o `inject` para fazer a soma.

```
class Array

  def soma
    self.inject(0, :+)
  end
end
```

Agora que definimos o método, podemos utilizá-lo como fazemos com qualquer método do array diretamente.

```
[].soma
[1, 2, 3].soma
```

No primeiro, teremos o retorno o e no segundo, 6 como o esperado.

A vantagem do uso do monkey patch é que podemos criar novos métodos para outras classes de nosso sistema às quais não temos acesso, como no caso as padrões do Ruby ou internas de gems. Inclusive o rails tem o Active Support que define diversas extensões para classes ruby, por exemplo, o método `#sum` é gerado pelo Active Support ao array.

Fique atento para não querer sair criando métodos para as classes ruby, pois você pode estar sobrescrevendo um método que existe, ou ainda um que já foi criado via monkey patch por uma gem como o Active Support. Utilize sabiamente o monkey patch.

O cenário só piora quando estamos utilizando `delegate`, que inclui alguns dos métodos do `Kernel`. Sendo assim, se o `rspec-expectations` for carregado antes do `delegate`, tudo irá funcionar corretamente, mas se o `delegate` for carregado primeiro, receberemos o erro.

Para a sintaxe de `should` funcionar, ela teria que ser definido em todos os objetos do sistema, mas o RSpec não possui todos os objetos do sistema e também não consegue garantir que isso ocorrerá sempre.

Se você, assim como eu, nunca recebeu nenhum erro devido a isso, sorte nossa! Esta nova sintaxe nos ajuda a continuar assim, com o RSpec funcionando sem gerar problemas enquanto escrevemos nossos testes.

Você deve se lembrar do `expect`, que aceitava apenas um bloco como parâmetro, como o caso a seguir.

```
it 'cria um novo pokemon' do
  expect do
    criador_pokemon.criar
  end.to change{ Pokemon.count }.by(1)
end
```

Com a nova sintaxe, o `expect` foi unificado. Utilizamo-lo passando apenas um parâmetro ou passando um bloco, como no exemplo anterior.

A pegadinha do sujeito implícito

Quando estamos utilizando matchers de uma linha, pode ficar meio verboso o uso da sintaxe de *expect*, pois teremos que passar o `subject` para o `expect` obrigatoriamente da seguinte maneira:

```
it { expect(subject).to be_a(ActiveRecord::Base)  }
```

Como pode ver, não ficou muito legal, pois uma das vantagens do teste de uma linha é podermos usar diretamente o matcher sem precisar definir o sujeito. É aí que vem a pegadinha: mesmo quando estamos utilizando a sintaxe de *expect*, o RSpec nos deixa utilizar o `should`.

```
it { should be_a(ActiveRecord::Base) }
```

Sendo assim, quando temos o sujeito implícito podemos continuar utilizando o `should`. Isso ocorre pois o `should`, quando usado no sujeito implícito, não depende do *monkey patch* no `Kernel`.

A partir do RSpec 3, além do `should` podemos utilizar uma nova sintaxe, o `is_expected.to`.

```
it { is_expected.to be_a(ActiveRecord::Base) }
```

O `is_expected.to` possui a sua contraparte, o `is_expected.to_not`. Dessa forma, em projetos novos utilizando o RSpec 3 prefira o uso do `is_expected.to`.

Vou começar um projeto novo: qual sintaxe usar?

Iniciaremos um projeto agora. Utilizaremos o RSpec mais recente, 2.xx (assumindo que ainda não tenhamos o RSpec 3 estável). Definimos com todo o time que iremos usar a nova sintaxe, para ser suave a nossa transição para o RSpec 3.

No entanto, começamos a perceber diversos `should` durante o código, afinal o time ainda está acostumado a utilizá-lo e acaba eventualmente esquecendo de usar o `expect`. Assim nossos testes ficam sem padrão nenhum, uns com uma sintaxe outros com outra.

Para resolver o problema, podemos dizer para o RSpec que queremos utilizar apenas a nova sintaxe. Para isso, adicionamos ao nosso `spec_helper.rb` a configuração `expect_with` e definimos apenas a sintaxe `expect`, passando o seguinte bloco:

```
RSpec.configure do |config|
  # ...
  config.expect_with :rspec do |c|
    c.syntax = :expect
  end
end
```

Deste modo, quando você mesmo ou alguém do time tentar utilizar a sintaxe do `should`, receberá um `NoMethodError` dado que o RSpec não fará o *monkey patch* para adicionar o `should`. Assim, todo o time utilizará apenas a nova sintaxe.

1.5 Descrevendo bem o seu teste

Devemos sempre ser claros no nosso teste, deixando explícito o que estamos testando. Sendo assim, mensagens de testes soltas e sem contexto não são uma boa prática. Vamos voltar ao teste do método `Pokemon#nome_completo`.

```
it 'exibe o nome e o id nacional' do
  pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
  expect(pokemon.nome_completo).to eq('Charizard - 6')
end
```

Nosso teste passa. No entanto, imagine um arquivo com dezenas de testes e você quer saber onde está o teste do `nome_completo`. Fica complicado de achar, dado que não há nenhuma referência a isto no nome do teste. Podemos melhorar a nossa leitura definindo o nome do método, com um bloco `describe`.

```
describe '#nome_completo' do
  it 'exibe o nome e o id nacional' do
    pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
    expect(pokemon.nome_completo).to eq('Charizard - 6')
  end
end
```

Observe que chamamos o nosso describe de `#nome_completo`. Utilizamos a tralha(#) para nos referenciarmos a métodos de instância e utilizamos o ponto(.) para nos referenciarmos a métodos de classe, assim como diz a convenção da documentação do Ruby. Dessa maneira, fica muito mais fácil de se entender o que está acontecendo em cada um dos testes.

De bônus, ao utilizarmos o formato de documentação do RSpec, temos uma saída bem legal e descritiva.

```
Pokemon
  #nome_completo
    exibe o nome e o id nacional
```

Para utilizarmos o formato de documentação, simplesmente passamos o parâmetro `--format d` da seguinte maneira ao RSpec: `rspec --format d`.

1.6 Não teste apenas o *happy path*

No nosso exemplo anterior, testamos apenas o *happy path*, o caminho mais comum de todos. Assumimos que todos os pokémons terão definidos o `nome` e o `id_nacional`, mas e quando um pokémon não possuir estes valores? Devemos testar estes casos também.

Para isso, criamos um novo pokémon sem definir nenhum dos valores e esperamos que o método nos retorne `nil`.

```
describe '#nome_completo' do
  it 'exibe o nome e o id nacional' do
    pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
    pokemon_2 = Pokemon.new
    expect(pokemon.nome_completo).to eq('Charizard - 6')
    expect(pokemon_2.nome_completo).to be_nil
  end
end
```

Para fazermos o nosso teste passar, simplesmente verificamos antes se o pokémon possui tais valores.

```
def nome_completo
  "#{nome} - #{id_nacional}" if nome && id_nacional
end
```

Observe, no entanto, que estamos fazendo 2 asserções dentro do mesmo teste, temos 2 `expect` no mesmo `it`. É uma boa prática definirmos apenas uma asserção por teste. Para fazermos isso, alteramos o nosso teste em 2: um que testa o pokémon que possui todos os valores e outro que testa o pokémon que não possui tais valores.

```
describe '#nome_completo' do
  it 'exibe o nome e o id nacional quando possui os valores' do
    pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
    expect(pokemon.nome_completo).to eq('Charizard - 6')
  end

  it 'é nil quando não possui o nome e o id nacional' do
    pokemon = Pokemon.new
```

```
    expect(pokemon.nome_completo).to be_nil
  end
end
```

Contextos para a melhor descrição

No nosso teste de `Pokemon#nome_completo`, vemos surgir a necessidade do uso de contexto. Observe que após o que realmente é esperado seguimos com a palavra "quando". Um bom modo de saber se precisamos de um contexto é quando vemos testes que possuem "se" (se o usuário está logado), "com" (com um usuário válido) e "quando". Pois cada um destes tem a sua contraparte, como, no nosso exemplo, o que possui os valores e o que não possui os valores.

Simplesmente refatoramos movendo a parte do contexto para um bloco `context` mantendo o nosso teste da seguinte maneira:

```
describe '#nome_completo' do
 context 'quando possui nome e o id nacional' do
    it 'exibe o nome e o id nacional' do
      # ...
    end
  end

  context 'quando não possui o nome e o id nacional' do
    it 'é nil' do
      # ...
    end
  end
end
```

Assim fica claro para todos do time que o método `#nome_completo` se comporta diferente baseado no que é definido no model `Pokemon`.

1.7 Definindo o sujeito

No exemplo anterior, estamos criando o objeto a ser testado dentro do próprio teste, no entanto é comum os contextos terem diversos testes. Podemos

resolver isso com uma variável de instância com o nosso objeto a ser testado, simplesmente criando `@pokemon` dentro de um bloco `before`.

```
context 'quando possui nome e o id nacional' do
  before do
    @pokemon = Pokemon.new(nome: 'Charizard', id_nacional: 6)
  end

  it 'exibe o nome e o id nacional' do
    expect(@pokemon.nome_completo).to eq('Charizard - 6')
  end
end
```

Se você utiliza o RSpec há algum tempo já deve ter feito algo assim, ou ter visto este padrão. No entanto, podemos fazer melhor utilizando o `let`.

Removemos o bloco `before` e utilizamos um bloco `let`, passando o nome do objeto que queremos como um símbolo no nosso caso `pokemon`.

```
context 'quando possui nome e o id nacional' do
 let(:pokemon) do
   Pokemon.new(nome: 'Charizard', id_nacional: 6)
 end

  it 'exibe o nome e o id nacional' do
    expect(pokemon.nome_completo).to eq('Charizard - 6')
  end
end
```

Observe que agora temos disponível o método `pokemon`, que foi criado pelo `let`, e não mais a variável de instância `@pokemon`.

Além disso, estamos utilizando a *DSL* do RSpec para o que ela foi criada. Temos a vantagem do `let` ser *lazy loaded*, ou seja, ele é criado na primeira vez que é chamado e cacheado até o final do teste que está sendo rodado. Se ele não for usado por algum teste, ele nem é criado, diferente de quando utilizamos o `before`.

Em casos em que precisamos garantir que o objeto seja criado antes do nosso teste, utilizamos o `let!`, que funciona como o `let`, com a diferença de que ele é criado antes do teste, e não quando é chamado como o `let`.

Isso não quer dizer que o `before` não tenha seu uso. Utilize-o para quando precisar criar o cenário do seu teste, mas não precisar da instância de nenhum dos objetos gerados. Como, por exemplo, criar 11 pokémons para testar uma paginação podemos criar todos eles dentro de um bloco `before` e testar apenas a quantidade dos pokémons retornados.

O tal do subject

No uso do padrão do bloco `before`, uma outra prática que surgiu foi o uso do `@it` para definir o objeto a ser testado.

```
context 'quando possui nome e o id nacional' do
  before do
    @it = Pokemon.new(nome: 'Charizard', id_nacional: 6)
  end

  it 'exibe o nome e o id nacional' do
    expect(@it.nome_completo).to eq('Charizard - 6')
  end
end
```

Afinal, o bloco `before`, além de definir o objeto a ser testado, pode estar gerando também outros objetos para montar o cenário do teste. Podemos alterar o nosso exemplo para utilizar o `subject`, simplesmente trocando o nosso `let` por `subject` e referenciando-nos a `subject` durante o nosso teste.

```
context 'quando possui nome e o id nacional' do
 subject do
   Pokemon.new(nome: 'Charizard', id_nacional: 6)
 end

  it 'exibe o nome e o id nacional' do
    expect(subject.nome_completo).to eq('Charizard - 6')
  end
end
```

Assim como o `let`, o `subject` também é *lazy loaded*. Você pode estar se perguntando qual a vantagem de se usar o `subject` em vez do `let`.

Uma das grandes virtudes do RSpec é a legibilidade. Imagine que tenhamos mais 2 `lets` compondo o nosso cenário. À primeira vista, não sabemos de antemão qual objeto está sendo testado, temos que olhar no teste. Utilizando o `subject` sabemos que aquele é o sujeito do nosso teste.

Se ainda preferirmos utilizar o `pokemon` diretamente, podemos passar um parâmetro para o `subject` definindo o nome do subject da seguinte maneira:

```
context 'quando possui nome e o id nacional' do

 subject(:pokemon) do
   Pokemon.new(nome: 'Charizard', id_nacional: 6)
 end

  it 'exibe o nome e o id nacional' do
    expect(pokemon.nome_completo).to eq('Charizard - 6')
  end
end
```

A vantagem neste caso é que, no meio de diversos `let`, sabemos exatamente qual o nosso sujeito.

Como tudo em programação, não há resposta correta ou apenas um caminho a se seguir. Há uma parcela da comunidade Ruby que apoia o uso da DSL [6] e outra que prefere o uso de métodos Ruby [2]. Utilize cada uma das abordagens e escolha qual seguir, baseado no contexto do seu projeto. Afinal, se o projeto já existe e estamos seguindo com um padrão, é melhor mantê-lo do que inserir uma abordagem diferente. Utilize novos projetos para experimentar o padrão diferente do que o habitual, e se não gostar. simplesmente mude.

1.8 No dia a dia não se esqueça de ...

utilizar sempre os matchers

Observe que, no nosso exemplo para testar se o nome completo do pokémon era `nil`, utilizamos o matcher `be_nil`.

```
it 'é nil' do
  expect(subject.nome_completo).to be_nil
end
```

Poderíamos ter utilizado o matcher `eq` passando o valor `nil` da seguinte maneira:

```
it 'é nil' do
  expect(subject.nome_completo).to eq(nil)
end
```

No entanto, é uma boa prática sempre utilizarmos os *matchers* que o RSpec nos oferece por questões de legibilidade.

Vale lembrar que ao utilizarmos a sintaxe *expect* não podemos utilizar o ==, dado que esta sintaxe não aceita os operadores diretamente. Resolvemos isso simplesmente com o matcher `eq`, que possui o mesmo comportamento.

Sendo assim, sempre que possível utilize os matchers do RSpec. Podemos ter uma lista dos matchers em: https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers.

não use o *should*

Uma prática que era comum da comunidade era a de todos os testes iniciarem com *should* ("deveria" em inglês).

```
it 'should have the name and the national_id' do
  expect(pokemon.full_name).to eq('Charizard - 6')
end
```

No nosso teste, temos em sua descrição algo como "deve ter o name e o national_id". O melhor modo de definirmos isto é utilizando a terceira pessoa do presente, alterando de *should* para *does*, da seguinte maneira:

```
it 'does have the name and the national_id' do
  expect(pokemon.full_name).to eq('Charizard - 6')
end
```

Se está começando um novo projeto, mas percebe que você e o time ainda estão colocando o *should*, mesmo que inconscientemente, utilize a gem

`should_not` (https://github.com/should-not/should_not) , que faz as specs que iniciarem com *should* falharem. Se o caso é um projeto antigo, podemos utilizar da gem `should_clean` (https://github.com/siyelo/should_clean) , que faz exatamente o que fizemos no nosso exemplo.

utilize um *coding style*

Defina com o seu time qual *coding style* utilizar, pois assim poderão evitar que cada teste esteja definido de uma maneira diferente. Utilize sempre os mesmos espaços, mesma quebra de linha etc. Uma simples convenção de qual coding style seguir ajuda a todos no time, já que os participantes se sentem sempre em casa, e não estranham como cada teste foi feito. Com um coding style bem definido e as dicas que vimos neste capítulo, fica muito mais fácil o uso do RSpec para todos do time.

O Better Specs [6] nos recomenda seguir o coding style definido na suíte de testes do mongoid (https://github.com/mongoid/mongoid) . Podemos utilizá-lo ou definir algo a partir dele, desde que todos do time concordem em utilizar o mesmo coding style.

Como o RSpec também é Ruby, recomendo sempre ficar de olho no *Ruby Style Guide* (https://github.com/bbatsov/ruby-style-guide) e no *Rails Style Guide* (https://github.com/bbatsov/rails-style-guide) , que possuem bastantes dicas legais de como definir o seu código.

Durante o livro, seguimos o coding style do Better Specs com algumas pequenas alterações para se adequar ao processo de impressão.

1.9 Conclusão

A diversão apenas começou, Vamos relembrar o que vimos neste comecinho do livro.

- Vimos as vantagens de escrever testes automatizados;
- Criamos nosso primeiro teste utilizando o Minitest;
- Criamos nosso primeiro teste utilizando o RSpec;
- Utilizamos a sintaxe *expect* do RSpec;

- Vimos que podemos utilizar o `should` quando estamos com o sujeito implícito;
- Vimos dicas de como escrever melhor os nossos testes;
- Não testamos apenas o *happy path*;
- Utilizamos `let` e `subject` para definirmos o cenário de nossos testes;
- Vimos dicas para usarmos no nosso dia a dia.

Continue aí, que no próximo capítulo veremos que testes que acessam a rede são um problema. Veremos como podemos contorná-los utilizando o **WebMock** e automatizando o processo com o uso do **VCR**.

Capítulo 2

Testes que acessam rede... WTF!?!

2.1 Introdução

Hoje em dia é comum termos que utilizar diversas APIs, como Twitter, Facebook etc. Temos que, por exemplo, obter todos os amigos de um usuário do Facebook, mas e para testar isso? Acessar a rede durante os testes nos dá muitos problemas:

- **Testes lentos**: se a cada um dos testes tivermos que acessar a rede, o processo ficará lento, e como reflexo nosso comportamento será começar a não querer escrever testes, para não aumentar o tempo de funcionamento da suíte de testes.

- **Testes quebradiços**: também chamados de falsos positivos, seriam aqueles que em alguns momentos passam e em outros não. Afinal, como estamos dependentes de rede, podemos ficar sem internet ou a API pode não responder, assim o teste às vezes passa, às vezes não.
- **Não poder testar sem rede**: quando estamos em um voo ou em lugares afastados dos grandes centros, pode haver momentos sem conexão à internet. Seria muito bom poder rodar nossa suíte de testes mesmo sem internet, mas se estivermos acessando rede não conseguiremos.

Não queremos nenhum destes problemas na nossa suíte. Vamos ver como podemos resolver isso.

2.2 Consumindo uma API

Temos o nosso model `Pokemon`, no entanto, não queremos ficar criando cada um dos pokémons manualmente. Para salvar os dados sobre todos os pokémons iremos utilizar a API pública do site http://pokeapi.co/.

Sabemos que cada pokémon possui um `id_nacional` em nossa API. Deste modo, pelo `id_nacional` do pokémon conseguimos pegar suas demais informações. Vamos implementar isso em nossa rails app.

O CriadorPokemon

Para iniciar, criaremos uma classe que recebe um `id_nacional` e cria um pokémon utilizando as informações vindas da API. Vamos primeiro ao teste:

```
describe CriadorPokemon do

  describe '#criar' do

    let(:criador_pokemon) do
      CriadorPokemon.new(6)
    end

    it 'cria um novo pokemon' do
```

```
      expect do
        criador_pokemon.criar
      end.to change{ Pokemon.count }.by(1)
    end
  end
end
```

Começamos fazendo apenas o teste para verificar se um pokémon foi criado. Vamos agora ao código para fazê-lo passar.

```
class CriadorPokemon

  def initialize(id_nacional)
  end

  def criar
    Pokemon.create
  end
end
```

Como se pode ver, não estamos consumindo a API ainda, afinal testamos apenas se criamos um pokémon e este teste passou. Agora vamos avançar mais um pouco e salvar o nome deste pokémon. Vamos ao teste.

```
describe 'pokemon criado' do

  before do
    criador_pokemon.criar
  end

  subject do
    Pokemon.last
  end

  it 'possui o nome correto' do
    expect(subject.nome).to eq('Charizard')
  end
end
```

Nosso teste diz que o nome do nosso pokémon deve ser *Charizard*, dado que estamos passando para ele o `id_nacional` 6. Agora vamos implementar o código para gerar o nome do pokémon.

Vamos primeiro criar o `initializer`, no qual atribuímos `id_nacional` a `@id_nacional`. O legal é que criamos um `reader` para podermos usar apenas `id_nacional` em nosso código, em vez de `@id_nacional` e, como definimos o reader como privado, não será possível acessar `id_nacional` publicamente. Isso, para nós, é o ideal, dado que o único objetivo desta classe é salvar um pokémon. Se fôssemos expor a interface `CriadorPokemon#id_nacional` publicamente, teríamos que escrever um teste para isto.

```
class CriadorPokemon

  def initialize(id_nacional)
    @id_nacional = id_nacional
  end

  private

    attr_reader :id_nacional
end
```

Vamos criar um método privado `endpoint`. A única coisa que ele faz é criar um objeto `URI` interpolando o endpoint da API com o nosso `id_nacional`. Como vamos utilizar o `Net::HTTP`, é uma boa definirmos um objeto `URI` para não precisarmos passar duas strings para o método `get`. Assim seguimos uma abordagem mais orientada a objeto pois temos um objeto que representa uma URI e não uma string. E de bônus ganhamos diversos métodos como `#hostname`, `#path` e qualquer outro que o `URI` nos oferecer.

```
def endpoint
  URI("http://pokeapi.co/api/v1/pokemon/#{id_nacional}/")
end
```

Agora sim vamos ao `cria_info`. Primeiro, usamos o `Net::HTTP` para fazer a nossa requisição GET e assim conseguimos a informação do

nosso pokémon. Em seguida, parseamos a resposta da requisição utilizando o `JSON.parse` e salvamos isto em `@info`.

```
private

  attr_reader :id_nacional, :info

  def cria_info
    resposta = Net::HTTP.get(endpoint)
    @info = JSON.parse(resposta)
  end
```

Nossa resposta é um JSON como algo assim:

```
{
  "attack": 84,
  "defense": 78,
  "moves": [
    {
      "learn_type": "machine",
      "name": "Dragon-tail",
      "resource_uri": "/api/v1/move/525/"
    },
  ],
  "name": "Charizard",
  "national_id": 6,
}
```

Removi diversas informações, pois isto é o suficiente para o nosso propósito. Caso queira visualizar todas as informações, basta acessar o endpoint http://pokeapi.co/api/v1/pokemon/6/.

Criamos o método `nome`, que nada mais é do que um reader do `info['name']`.

```
def nome
  info['name']
end
```

Alteramos nosso initializer para criar a informação do pokémon.

```
def initialize(id_nacional)
  @id_nacional = id_nacional
  cria_info
end
```

Finalizamos alterando o create para salvar o pokémon com o name correto.

```
def criar
  Pokemon.create(nome: nome)
end
```

Nossos testes agora estão verde, todos os testes passam, no entanto estamos enfrentando todos os problemas mencionados no começo do capítulo.

2.3 WebMock ao resgate

O `WebMock` é uma biblioteca que nos permite fazer stub e asserções de requisições HTTP, exatamente o que precisamos para resolver o nosso atual problema.

Instalação

Simplesmente adicionamos ao nosso `Gemfile` a seguinte linha e rodamos `bundle`.

```
gem 'webmock'
```

Em seguida, adicionamos ao `spec_helper.rb` a linha.

```
require 'webmock/rspec'
```

Forjando uma requisição HTTP

Antes de continuarmos, vamos rodar os nossos testes. WTF?! Mas agora eles quebram, deixando a seguinte mensagem:

```
WebMock::NetConnectNotAllowedError:
 Real HTTP connections are disabled.
 Unregistered request:
 ...
```

Este é o WebMock nos avisando que estamos tentando fazer uma requisição HTTP e no corpo da mensagem do erro ele ainda nos ensina a corrigi-lo utilizando o método `stub_request`.

Uma das funcionalidades do WebMock é garantir que nenhuma requisição HTTP seja feita em nossos testes, ou seja, ele nos alerta sobre aqueles problemas que vimos no começo e nos dá uma dica de como resolvê-los utilizando o WebMock.

No nosso caso, não é importante quais os parâmetros que estamos passando no header, dado que a API não se importa com isso. O que queremos forjar é apenas o retorno da API. Vamos seguir a dica do WebMock e utilizar o `stub_request`. Antes dos nossos testes de create, adicionamos a chamada ao `stub_request`.

```
describe '#criar' do

  before do
    stub_request(:get, 'http://pokeapi.co/api/v1/pokemon/6/')
      .to_return(status: 200, body: '', headers: {})
  end
  # ...
end
```

Agora estamos forjando a resposta. Toda requisição `GET` feita à URL http://pokeapi.co/api/v1/pokemon/6/ terá o mesmo valor retornado. Vamos rodar o teste e ver se está verde. Ainda estamos com erros, mas desta vez temos o seguinte:

```
TypeError:
  no implicit conversion of nil into String
```

Muito bom, agora o erro é no nosso código e não devido ao acesso à rede. Verificando, conseguimos descobrir que esse erro ocorre no método `cria_info` exatamente em `JSON.parse(resposta)`, pois neste ponto forjamos a requisição, ou seja o valor da resposta. Mas como definimos o `body` como vazio, o `JSON.parse` não consegue fazer o parse de `nil` e retorna `TypeError`.

Vamos arrumar a nossa resposta e dar um valor ao `body`. Para isso, acessamos http://pokeapi.co/api/v1/pokemon/6/, salvamos apenas o que interessa para a nossa API, e passamos isto para o WebMock.

```
describe '#criar' do

  before do
    body = '{' \
      ' "name": "Charizard"' \
      '}'
    stub_request(:get, 'http://pokeapi.co/api/v1/pokemon/6/')
      .to_return(status: 200, body: body, headers: {})
  end

  # ...
end
```

Rodamos os testes, tudo está verde e nossos testes, passando.

Forjando like a pro

No exemplo anterior, conseguimos forjar a nossa requisição e agora não estamos mais enfrentando os problemas detalhados no início da seção. No entanto, enfrentamos os seguintes problemas:

- **Processo factível a erro:** se continuarmos criando nossas respostas, estamos sujeitos a eventualmente escrever algo errado, como, por exemplo, definir um valor que seria uma string como inteiro. Isso pode nos levar a problemas inesperados.
- **Resposta incompleta:** no exemplo anterior, forjamos apenas os campos que estamos usando no momento. Ou seja, se precisarmos de novos campos, teremos que acessar a API mais uma vez manualmente para obtermos os dados e adicioná-los ao nosso teste.

2.4 Utilizando o cURL

O cURL é um utilitário de linha de comando para transferência de dados que entende vários protocolos. Ou seja, podemos acessar uma URL via cURL e

obter sua resposta. Por padrão, ele já vem instalado no Ubuntu e no Mac OS X.

Instalando o cURL no Windows

Acesse a página de download do cURL em http://curl.haxx.se/dlwiz/

1) **Selecione o tipo de pacote**: curl executable
2) **Selecione o sistema operacional**: Windows / Win32 ou Win64
3) **Selecione o *flavour***: Generic
4) **Selecione a versão**: Unspecified
5) **Faça o download**: simplesmente clicando no botão de download

Extraia o arquivo zip que acabou de baixar e mova-o para o `windir` (por padrão C:\WINDOWS). Abra o prompt de comando e execute `$ curl --version`, você receberá a versão do cURL instalada.

Podemos utilizar o cURL da seguinte maneira no terminal.

```
$ curl http://pokeapi.co/api/v1/pokemon/6/
```

Ele exibirá o body da resposta HTTP direto no seu terminal. Se você manja um pouco mais de Linux, sabe que podemos redirecionar o output do terminal para um arquivo utilizando o `>` e, para facilitar a nossa vida mais ainda, o `WebMock` aceita um arquivo como resposta. Vamos alterar nossa spec para utilizar uma resposta vinda do cURL.

Primeiro, criamos um arquivo de fixture em `spec/fixtures/services/criador_pokemon/resposta.txt`, que irá armazenar a nossa resposta. Depois, salvamos nossa resposta do servidor utilizando o cURL da seguinte maneira:

```
$ curl -is http://pokeapi.co/api/v1/pokemon/6/ > \
spec/fixtures/services/criador_pokemon/resposta.txt
```

A opção `i` que passamos para o cURL diz para ele incluir os cabeçalhos na resposta. Dado que o `Webmock` entende a resposta do cURL, não teremos que forjar o corpo nem os cabeçalhos da resposta.

Já a opção `s` é para utilizarmos o modo silencioso, ou seja, não receberemos o output direto no terminal.

Agora que temos a nossa resposta salva, temos que alterar nosso teste para utilizar este arquivo.

```
describe '#criar' do
  before do
    caminho_arquivo =
      'spec/fixtures/services/criador_pokemon/resposta.txt'
    arquivo_resposta = File.new(caminho_arquivo)
    stub_request(:get, 'http://pokeapi.co/api/v1/pokemon/6/')
    .to_return(arquivo_resposta)
  end

  # ...
end
```

Rodamos nossos testes e continua tudo verde, como esperado. Se por algum motivo a resposta da API mudar, simplesmente rodamos o cURL para atualizarmos a nossa resposta.

2.5 Mas eu quero automatizar isso...

Vimos que com o Webmock resolvemos os problemas relacionados ao uso de rede em nossos testes. Aprendemos que podemos usar o cURL para forjar melhor nossas requisições ao utilizarmos o `WebMock`.

Talvez você tenha se sentido inclinado a criar um helper, que lhe ajude a automatizar este processo de utilização do cURL em conjunto com o WebMock. Se sim, é melhor parar por aí, afinal já temos o VCR, que veremos adiante — uma gem que faz o uso do WebMock agilizando o processo da criação das fixtures, além de outras coisas legais.

2.6 VCR??? É O VIDEOCASSETE DE QUE MEU PAI FALA?

Se você viveu nos anos 80 e 90 com certeza se lembrará do videocassete, o ancestral do DVD e do Blu-ray. Da época em que a única coisa que se tinha ao alugar um filme era o próprio filme, em que se tinha que rebobinar a fita ao entregar na locadora para não pagar multa, da época em que ainda existiam locadoras.

O VCR que iremos utilizar é uma biblioteca ruby que utiliza bibliotecas de stub de requisições HTTP, como WebMock, para criar as fixtures, aqui chamadas de cassetes. Assim, automatizamos o processo de criação das fixtures.

Instalação

Simplesmente, adicionamos ao nosso `Gemfile` a seguinte linha e rodamos `bundle`:

```
gem 'vcr'
```

Em seguida, criamos um arquivo em `spec/support/vcr.rb` com o seguinte conteúdo.

```
VCR.configure do |c|
  c.cassette_library_dir = 'spec/fixtures/vcr_cassettes'
  c.hook_into :webmock
end
```

A configuração `cassette_library_dir` que fazemos é para definir onde armazenaremos nossos cassetes. Em seguida, definimos o `hook_into`, que diz qual será a biblioteca de stub que usaremos, e definimos o WebMock.

2.7 UTILIZANDO O VCR

Removendo o WebMock

Antes de iniciarmos o uso do VCR, devemos remover o WebMock de nosso teste.

Primeiro, removemos a nossa fixture:

```
$ rm spec/fixtures/services/criador_pokemon/resposta.txt
```

E removemos a chamada do `stub_request` no nosso teste:

```
describe CriadorPokemon do

  describe '#criar' do

    let(:criador_pokemon) do
      CriadorPokemon.new(6)
    end

    # ...
  end
end
```

Vamos agora rodar nossos testes e ver o que acontece. Recebemos como resposta algo como:

```
Failure/Error: CriadorPokemon.new(6)
VCR::Errors::UnhandledHTTPRequestError:


  ================================================================
  An HTTP request has been made that VCR does not know how
  to handle:
    GET http://pokeapi.co/api/v1/pokemon/6/

  There is currently no cassette in use. There are a few ways
  you can configure VCR to handle this request:

...
```

Omiti o restante do texto para focarmos no que interessa aqui para nós. Assim como o WebMock, ele nos notifica ao tentarmos acessar rede em nossos teste.

Aplicando o VCR

Agora que removemos o WebMock, vamos aplicar o VCR. O VCR utiliza de uma sintaxe diferente do WebMock: ele aceita um bloco como parâmetro e todo o acesso à rede que houver dentro do bloco é salvo em um arquivo de fixture, tanto a requisição como a resposta. Nosso teste fica da seguinte maneira:

```
describe '#criar' do

  let(:criador_pokemon) do
    CriadorPokemon.new(6)
  end

  it 'cria um novo pokemon' do
    expect do
      VCR.use_cassette('CriadorPokemon/criar') do
        criador_pokemon.criar
      end
    end.to change{ Pokemon.count }.by(1)
  end

  describe 'pokemon criado' do

    before do
      VCR.use_cassette('CriadorPokemon/criar') do
        criador_pokemon.criar
      end
    end
    # ...
  end
end
```

Utilizamos o método `VCR.use_cassette` para definir a nossa fixture utilizando o VCR, e passamos 2 parâmetros: o primeiro é o nome do arquivo de fixtures que queremos; o segundo, um bloco com a ação que acessa rede em nosso teste — no nosso caso, a chamada ao `criador_pokemon.criar`.

O VCR salva as fixtures utilizando o formato `YAML`, salvando o arquivo no caminho que definimos na nossa configuração em conjunto com o que

definimos no `VCR.use_cassette`.

Sendo assim, após rodarmos os nossos testes, teremos um novo arquivo criado. Este é criado apenas na primeira requisição; nas demais já utilizamos o arquivo de fixture. O arquivo é salvo em `spec/fixtures/vcr_cassettes/CriadorPokemon/criar.yml`, com algo como:

```
---
http_interactions:
- request:
    method: get
    uri: http://pokeapi.co/api/v1/pokemon/6/
    body:
      encoding: US-ASCII
      string: ''
    headers:
      ...
  response:
    status:
      code: 200
      message: OK
    headers:
      ...
    body:
      encoding: UTF-8
  ...
```

Omiti boa parte do `YAML`, mas o trecho apresentado é interessante para vermos que ele salva a requisição e a resposta no mesmo arquivo, utilizando o formato `YAML`.

Como se pode ver, reduzimos bastante o nosso trabalho. Agora não precisamos mais criar o arquivo manualmente, já que delegamos isto para o VCR, e ele ainda nos avisa quando esquecemos de algum teste que por um acaso esteja acessando rede. Neste caso simplesmente adicionamos o VCR e o resolvemos.

Blocks, blocks everywhere

A sintaxe que estamos utilizando até o momento é a padrão do VCR. O problema é que ela pode parecer bastante verbosa e confusa, principalmente quando está em conjunto com os blocos do RSpec. Como no nosso exemplo:

```
it 'cria um novo pokemon' do
  expect do
    VCR.use_cassette('CriadorPokemon/criar') do
      criador_pokemon.criar
    end
  end.to change{ Pokemon.count }.by(1)
end
```

Acabamos com três blocos aninhados: o do teste, o do matcher do RSpec e o do VCR.

Conheça o metadado do RSpec

Como o Ruby é uma linguagem que preza por legibilidade, podemos configurar o VCR para utilizar o metadado do RSpec.

Primeiro, alteramos o nosso config do VCR em `spec/support/vcr.rb`:

```
VCR.configure do |c|

  # ...
  c.configure_rspec_metadata!
end
```

Também alteramos o RSpec para tratar símbolos como se possuíssem o valor `true`, modificando o `spec/spec_helper.rb`:

```
RSpec.configure do |config|

  # ...
  config.treat_symbols_as_metadata_keys_with_true_values = true
end
```

Agora vamos alterar nosso teste para usar a nova sintaxe.

```
describe CriadorPokemon do

  describe '#criar', :vcr do

    # ...
  end
end
```

Removemos todos os blocos do nosso código e adicionamos apenas o símbolo `:vcr` na descrição do nosso método create. Se não configurássemos o RSpec com o `treat_symbols_as_metadata_keys_with_true_values`, teríamos que definir o valor de `:vcr` da seguinte maneira: `vcr: true`.

Rode os testes e veja que tudo continua passando. Só que agora com menos blocos, e uma sintaxe mais elegante.

Cassetes, Cassetes everywhere

Após rodar os testes, verificarmos que foi criado 1 arquivo de fixture para cada bloco `it`, cada teste. Ou seja, para cada teste, pelo menos 1 vez foi acessada a rede e criada uma fixture. O problema é se alterarmos o nosso teste, mudando o endpoint por exemplo. Teríamos que recriar todas estas fixtures se continuássemos seguindo esta abordagem. Dado que sempre estamos fazendo a mesma requisição, o ideal seria definirmos 1 arquivo de fixture apenas, como quando estávamos utilizando o bloco do VCR.

Para resolver isso, primeiro apagamos todas as fixtures novas criadas. Em seguida, alteramos o nosso teste.

```
describe CriadorPokemon do

  describe '#criar', vcr:
        { cassette_name: 'CriadorPokemon/criar' } do

    # ...
  end
end
```

Simplesmente passamos um paramêtro `cassette_name` para o VCR,

de modo que, em todos os testes daquele bloco, ele usará o mesmo cassete que é o que queremos neste caso. Mas haverá casos em que usaremos um cassete diferente por contexto, por exemplo.

Na nossa hipótese, apontamos para o mesmo cassete que já havíamos criado, no entanto, se não houvesse nenhum cassete para a requisição, ele seria criado no primeiro teste e utilizado nos seguintes.

2.8 Dados sensíveis no VCR

Observe o cassete gerado pelo VCR, especificamente na parte de requisição.

```
---
http_interactions:
- request:
    method: get
    uri: http://pokeapi.co/api/v1/pokemon/6/
    body:
      encoding: US-ASCII
      string: ''
    headers:
      Accept-Encoding:
      - gzip;q=1.0,deflate;q=0.6,identity;q=0.3
      Accept:
      - '*/*'
      User-Agent:
      - Ruby
      Host:
      - pokeapi.co
```

Perceba que todos os dados que passamos como `uri`, `body` e os `headers` são salvos neste arquivo.

Enquanto estamos lidando com projetos privados e APIs públicas, não precisamos nos preocupar com o vazamento da informação, dado que ela está limitada a um número de pessoas.

No entanto, vamos inverter a situação: e se estivermos trabalhando em um projeto open source que tenha que acessar uma API privada? Acessar a rede não é uma opção durante os testes, mas também não seria legal mantermos nossas credenciais disponíveis para qualquer um no github.

Filtrando dados sensíveis

Vamos mudar um pouco agora o nosso cenário. Dado que a nossa API anterior é publica, vamos agora utilizar uma API privada, a API do moip (http://moip.com.br) .

O moip possui uma API para o seu chamado checkout transparente. Vamos abstrair detalhes e focar apenas no que nos interessa: devemos realizar um POST para a seguinte URI https://MOIP_TOKEN:MOIP_KEY@desenvolvedor.moip.com.br/sandbox/ws/alpha/EnviarInstrucao/Unica.

> **O CHECKOUT TRANSPARENTE DO MOIP**
>
> Em diversos serviços de intermediador de pagamento, PagSeguro e PayPal, por exemplo, temos um fluxo mais ou menos assim.
>
> 1) Os produtos são adicionados ao carrinho no e-commerce;
> 2) Ao clicarmos em comprar somos direcionados ao intermediador do pagamento;
> 3) Finalizando o pagamento, retornamos ao e-commerce com uma mensagem de sucesso.
>
> O checkout transparente remove o segundo passo. Nós não somos direcionados ao intermediador de pagamento, todo o processo é feito dentro do próprio e-commerce.
>
> Deste modo, utilizamos um intermediador de pagamento, mas o usuário final não tem esta percepção, ficando o processo transparente para ele.

Perceba que o endpoint do moip necessita que passemos o nosso moip token e nossa moip key em cada uma das requisições. E como vimos, o VCR irá salvar esses dados na fixture.

Para resolver isso, primeiro temos que dizer ao VCR quais dados são considerados sensíveis. Isso é feito com uma configuração da seguinte maneira:

```
VCR.configure do |c|

  # ...
  c.filter_sensitive_data('<MOIP_KEY>') { 'my_key' }
  c.filter_sensitive_data('<MOIP_TOKEN>') { 'my_token' }
end
```

Desta forma, sempre que o VCR encontrar `my_key`, ele substituirá no cassete para `<MOIP_KEY>`.

Com isto configurado, todos os nossos cassetes que usarem `my_key` e `my_token` serão substituídos. A nossa requisição ao endpoint do moip seria salva assim:

```
---
http_interactions:
- request:
    method: post
    uri: https://<MOIP_DEV_KEY>:<MOIP_DEV_TOKEN>@desenvolvedor \
    .moip.com.br/sandbox/ws/alpha/EnviarInstrucao/Unica
```

Agora nossos cassetes não possuem mais nenhuma informação do moip token e key reais. No entanto, se olharmos com atenção, esta informação não se encontra mais nos cassetes e, no entanto, ela ainda é disponível na configuração do VCR.

Para removermos por completo o valor hardcoded de moip token e key de nosso projeto, podemos utilizar o `dotenv`, que nos permite criar variáveis de ambiente no escopo da nossa aplicação.

DOTENV

Armazenar as configurações da app em variáveis de ambiente é um dos princípios do *twelve-factor app*(http://12factor.net) . Qualquer configuração que mude dependendo do ambiente (development, staging, production etc.) deve ser armazenada em uma variável de ambiente como:

- Dados de banco de dados, Memcached etc.
- Dados de serviços externos como Amazon S3 ou Facebook.

O `dotenv` carrega as variáveis definidas no arquivo `.env` na raiz de sua app e torna disponível o valor das variáveis na constante `ENV`, facilitando o processo de se definir variáveis de ambiente apenas para o escopo do projeto.

Criadas as variáveis de ambiente, simplesmente alteramos nossa configuração para:

```
VCR.configure do |c|

  # ...
  c.filter_sensitive_data('<MOIP_KEY>') { ENV['MOIP_KEY'] }
  c.filter_sensitive_data('<MOIP_TOKEN>') { ENV['MOIP_TOKEN'] }
end
```

E a definimos normalmente no nosso arquivo `.env` como solicitado pelo `dotenv`. Vale lembrar que não adicionamos este arquivo ao repositório, assim garantimos que apenas nós temos tais credenciais. Não se esqueça de deixar claro no seu projeto open source que, para o uso correto, deve-se criar um arquivo `.env` e preencher tais variáveis.

2.9 URIs não determinísticas

Nos exemplos anteriores, sempre estávamos acessando a mesma URI em nosso teste, mas em alguns momentos podemos não ter total controle disto.

Imagine a seguinte situação: estamos utilizando o Paperclip (Gem de upload de arquivos) e dizemos para ele que, no nosso model Jogo, é obrigatório o campo `capa`, que é uma imagem do Paperclip. Vamos a ele.

```
class Jogo < ActiveRecord::Base

  # ...

  validates :capa, attachment_presence: true

  has_attached_file :capa,
                    storage: :s3,
                    bucket: ENV['AMAZON_S3_BUCKET'],
                    path: "jogos/capas/:id.:style.:extension",
                    s3_credentials: {
               access_key_id: ENV['AMAZON_ACCESS_KEY_ID'],
               secret_access_key: ENV['AMAZON_SECRET_ACCESS_KEY']
                    }
end
```

Configuramos o Paperclip para mandar nossas imagens para a Amazon S3, e definimos um caminho dos arquivos salvos. Utilizamos o id do nosso model para geramos a imagem.

A princípio não teremos problemas nenhum, a não ser se chegarmos a uma ação em que não possamos definir o valor de `id` do model Jogo de antemão. Um destes casos é uma simples action de `create` em um controller de uma app rails.

Action create

Vamos agora fazer uma simples action de create para o nosso model game.

```
class JogosController < ApplicationController

  # ...

  def create
    @game = current_user.jogos.build(params[:jogo])
    if @game.save
```

```
      redirect_to new_jogo_path
    else
      render 'new'
    end
  end
end
```

Trata-se de uma simples action comum em diversas app rails por aí. Vamos dar uma olhada nos testes.

```
describe JogosController do

  describe "POST 'create'" do

    context 'com sucesso' do

      let(:params) do
        {
          'game' => {
            'name' => 'Zelda',
            'platform' => 'wii',
            'capa' => fixture_file_upload('/lorempixel.png')
          }
        }
      end

      it 'cria um novo jogo'  do
        expect do
          post :create, params
        end.to change(Jogo, :count).by(1)
      end
    end
  end
end
```

É um teste bem comum em diversas rails apps também. Mas atente ao detalhe de que utilizamos o `fixture_file_upload`, que é um helper que nos ajuda a enviar arquivos em nossos testes de controller.

Ao rodarmos os testes, deparamo-nos com a mensagem do VCR dizendo

que estamos fazendo uma requisição e que ele não sabe como tratar. Neste caso, quem está fazendo a requisição é o Paperclip para enviar nossa imagem para a Amazon S3. Como fizemos anteriormente, adicionamos o VCR.

```
describe JogosController do

  describe "POST 'create'", :vcr do

  # ...
end
```

Rodamos o teste novamente e está tudo verde. No entanto, agora vem a pegadinha: rode os testes novamente. Uou! Recebemos a mensagem de erro novamente do VCR, mas agora já temos o bloco do VCR e uma fixture criada! O que pode estar errado?

Request matching

O VCR utiliza as requisições para fazer um match e saber qual resposta retornar. Por padrão, utiliza-se o verbo HTTP e a URI, sendo assim, se seu teste fizer requisições diferentes a cada vez que rodar, o VCR retornará um erro informando que não sabe lidar com isso.

Vamos entender em que momento estamos fazendo esta requisição diferente. Olhando novamente para o nosso setup do Paperclip, deparamo-nos com a configuração do `path`:

```
path: "jogos/capas/:id.:style.:extension"
```

Como estamos testando uma action de create, o nosso `id` é dinâmico: a cada vez que rodamos os testes é gerado um `id` diferente e, como consequência, a URI que o Paperclip acessa para enviar o arquivo para a Amazon S3 também é diferente, dado que é passado este `path`.

Abrindo o cassete do VCR, vemos a URI do request algo como https://myproject-development.s3.amazonaws.com/jogos/capas/2.original.png.

Ao rodarmos o teste novamente, o 2 do `2.original.png` pode não ser mas 2, mas sim qualquer id gerado pelo banco de dados.

Resolvendo o problema

Procurando na documentação do VCR, podemos nos deparar com o modo de gravação `:new_episodes`. No entanto, ele não resolve nosso problema: uma vez que a requisição não deu match, ele adiciona um novo cassete no mesmo arquivo, ou seja, no nosso caso, seria como se não estivéssemos usando do VCR. Cada vez que rodássemos os testes seria considerada um novo episódio.

Um método que quase resolve o nosso problema é o `VCR.request_matchers.uri_without_param`. Contudo, ele funciona somente com parâmetros passados via query string.

Poderíamos resolver isso criando um matcher customizado. Mas para este caso em específico seria esforço demais sem necessidade, afinal sabemos que o Paperclip funciona e é bem testado. Queremos apenas que ele não quebre nosso teste de controller e que não tenhamos que testar o Paperclip novamente.

Para isso alteramos o novo VCR para que o `body` seja usado no macthing de requisição.

```
describe JogosController do

  describe "POST 'create'",
      vcr: { match_requests_on: [:body] } do

  # ...
end
```

Deste modo, não estamos nos importando com o método HTTP e a URI da requisição. Apenas estamos enviando o mesmo `body` sempre, ou seja, a imagem.

Assim conseguimos fazer com que nossos testes continuem passando, e não precisamos testar novamente o Paperclip no teste do controller. Somente precisamos verificar que ele não nos gera problemas devido a URIs não determinísticas.

Vale lembrar que se fosse um código nosso que estivesse acessando a URI não determinística, e não o Paperclip, faria sentido utilizarmos um matcher customizado, dado que estaríamos testando a requisição.

2.10 Conclusão

Neste capítulo vimos diversas dicas e técnicas que nos ajudam quando estamos escrevendo testes que acessam rede, entre elas:

- Testes que acessam rede podem ser um grande problema;
- Construímos uma simples classe que acessa a http://pokeapi.co/;
- Utilizamos o WebMock para forjar a nossa resposta HTTP;
- Forjamos nossa resposta HTTP de uma melhor maneira usando o cURL em conjunto com o WebMock;
- Automatizamos o processo de forjar respostas HTTP utilizando o VCR;
- Configuramos o VCR para integrar-se ao RSpec utilizando o metadado do RSpec;
- Diminuímos o número de fixtures geradas ao utilizarmos o metadado do RSpec;
- Vimos como filtrar dados sensíveis da fixture do VCR;
- Aprendemos a utilizar o VCR em URIs não determinísticas, como ao enviar um arquivo para a Amazon S3.

No próximo capítulo veremos como podemos nos livrar das fixtures de uma app rails utilizando a *factory_girl*. Se já utiliza a factory_girl não se preocupe pois haverá bastante coisa legal como utilizá-la para objetos que não são Active Record e testes para ela.

Capítulo 3

Fixtures são tão chatas! Conheça a factory_girl

3.1 Introdução

Se em algum momento já trabalhamos em alguma rails app, utilizamos fixtures ou factories para testar. Se você já a conhece, pode continuar lendo, porque tem bastante conteúdo legal, mesmo pra quem já usa a factory_girl no dia a dia! Se nunca usou a factory_girl, vou convencê-lo agora! =)

Provavelmente tem alguma fixture assim no seu código.

```
charizard:
  nome: Charizard
  id_nacional: 6
  ataque: 89
```

E um código que a utiliza da seguinte maneira.

```
describe '#nome_completo' do
  it 'exibe o nome e o id nacional' do
    pokemon = pokemons(:charizard)
    expect(pokemon.nome_completo).to eq('Charizard - 6')
  end
end
```

O problema desta abordagem é que, no nosso teste, não fica claro quais são os valores definidos para o objeto pokémon. Qual é o ataque deste pokémon? Qual o seu id_nacional? Para descobrirmos tudo isso, temos que abrir o nosso arquivo de fixture e ver tais informações, ou seja, a fixture é uma forma de *Mystery Guest* devido à sua definição e está distante do contexto em que estamos utilizando.

Claro que podemos trocar a fixture ali para usar diretamente o Active Record com `Pokemon.create`; deste modo teremos a definição junto ao contexto. Mas esta abordagem não é nem um pouco prática, pois ao adicionarmos um novo campo com validação no model `Pokemon`, teremos que ir atrás de diversas chamadas a `Pokemon.create` para corrigir os erros de validação que serão lançados e, por consequência, fazem nossos testes quebrarem. Além disso, estaríamos definindo diversos campos que não têm nada a ver com o contexto que estamos testando apenas para a validação do Active Record passar.

Como nem tudo são flores as factories são lentas, devido a sempre estarem persistindo no banco de dados, mas ganham em legibilidade e flexibilidade. Não se preocupe, veremos algumas dicas de como evitar esta lentidão neste capítulo.

Temos estas duas vertentes na comunidade: os que usam factory_girl [7] e os que usam fixtures [4]. Leia este capítulo, comece um novo projeto e utilize factories, faça a sua própria experiência.

3.2 Instalação

Simplesmente adicionamos a gem ao nosso `Gemfile` e, em seguida, rodamos `bundle`.

```
group :development, :test do
  gem 'factory_girl_rails'
end
```

Se não estivermos utilizando Rails, podemos adicionar a gem `factory_girl` no lugar de `factory_girl_rails`.

Eu costumo utilizar a `factory_girl` no ambiente de development e test pois assim consigo usar as factories no console de uma app rails.

3.3 Criando nossa primeira factory

Para nossa primeira factory, criamos um arquivo em `spec/factories/usuarios.rb`, iniciamos com o bloco `FactoryGirl.define` e, depois, chamamos o método `factory`, que recebe um bloco com a definição da nossa factory. Criamos uma factory de `usuario`, que é mapeado para o nosso model `Usuario`.

```
FactoryGirl.define do
  factory :usuario do
    nome 'Mauro'
    email 'mauro@helabs.com.br'
  end
end
```

A factory_girl carrega automaticamente as factories em `spec/factories.rb` e `spec/factories/*.rb`. Como boa prática, criamos um arquivo de factory para cada model – no nosso caso aqui, o `usuarios.rb`.

Uma boa prática para quando estamos definindo uma factory é definir apenas os atributos que são necessários ao model, ou seja, aqueles que são obrigatórios devido à validação do Active Record. Veremos como criar factories mais especificas adiante.

3.4 Utilizando a factory

Definimos a nossa factory no exemplo anterior. Vamos brincar com ela um pouco direto pelo console, de preferência em modo sandbox. Para isso,

`rails c --sandbox`. Assim, todos os registros que inserirmos no banco de dados serão apagados ao sairmos do console.

Para criar nossa primeira factory usamos.

```
FactoryGirl.create(:usuario)
```

Como retorno, teremos um objeto Active Record que foi salvo no nosso banco de dados, utilizando os valores que definimos na nossa factory na seção anterior com o bloco `FactoryGirl.define`.

Utilizando `FactoryGirl.create(:usuario)`, novamente receberemos um erro de validação, dado que no nosso model é definido que o campo `email` deve ser único. Agora é que vem uma das vantagens da factory em relação à fixture: podemos alterar a nossa factory sem a necessidade de criar uma nova. Fazemos da seguinte maneira:

```
FactoryGirl.create(:usuarios, email: 'mauro@helabs.com.br')
```

O registro foi criado com sucesso. A factory_girl nos deixa passar como parâmetro cada um dos campos definidos no Active Record. Podemos alterar a nossa factory sem a necessidade de criarmos outra factory no `spec/factories/usuarios.rb`.

3.5 Factories nos testes

No exemplo anterior, vimos como utilizar a `FactoryGirl` no console para apenas conhecermos seu comportamento. O seu uso real é no ambiente de testes. Vamos a ele.

Uma simples action show

Vamos supor que temos uma action `show`, que exibe o conteúdo de um artigo. Um possível teste seria verificarmos se a variável `@artigo` foi atribuída corretamente para a nossa view. Para isso, criaremos uma instância de `Artigo` utilizando a factory_girl e nosso teste simplesmente verifica se foi passado para a view o `@artigo`, pelo método `assigns`.

Veja o teste.

```
describe "GET 'show'" do
  let!(:artigo) do
    FactoryGirl.create(:artigo)
  end

  before do
    get :show, id: artigo
  end

  it 'assigns a artigo' do
    expect(assigns(:artigo)).to eq(artigo)
  end
end
```

Vamos agora ao controller.

```
class ArtigosController < ApplicationController

  def show
    @artigo = Artigo.find(params[:id])
  end
end
```

Rodamos o teste e agora ele está passando. Seguindo o TDD (vermelho, verde e refatorar), estamos no momento de refatorar. No nosso controller, não tem muito o que fazer, está bem legal para o momento. Mas podemos melhorar o nosso teste, mais especificamente, o modo como usamos a `FactoryGirl`.

Configurando a FactoryGirl

No nosso exemplo anterior, utilizamos `FactoryGirl.create(:artigo)`. Para usar a nossa factory de `Artigo`, no entanto, é bastante verboso ter que sempre se referir à classe `FactoryGirl` quando queremos criar uma factory.

Por isso, a factory_girl nos oferece a opção de usarmos apenas `create` em nossos testes. Basta adicionarmos ao nosso `spec_helper.rb` o seguinte:

```
RSpec.configure do |config|
  # ...
  config.include FactoryGirl::Syntax::Methods
end
```

Assim podemos alterar nosso `let!` que usa a factory para:

```
let!(:artigo) do
  create(:artigo)
end
```

Agora temos uma versão bem menos verbosa no uso de nossa factory.

Uma simples action create

Em um teste de controller da action de create de uma rails app qualquer, provavelmente teremos um trecho como a seguir.

```
describe "POST 'create'" do
  let(:params) do
    {
      artigo: {
        titulo: 'Meu título',
        conteudo: 'Conteúdo do artigo'
      }
    }
  end

  it 'create a new artigo' do
    expect do
      artigo :create, params
    end.to change(Artigo, :count).by(1)
  end
end
```

A declaração de `params` está bem grandinha, dado que temos que montar o hash com os atributos. Neste caso, passamos apenas dois atributos, mas em um model maior a situação só pioraria.

Como estamos utilizando a factory_girl já definimos estes valores em nossa factory, dado que são os obrigatórios. No entanto, o `create` nos retorna um objeto Active Record e cria um registro no banco, e não é isso que queremos. Queremos apenas os valores que passamos para a factory, para isso utilizamos o método `attributes_for`. Vamos alterar o nosso `let`.

```
let(:params) do
  { artigo: attributes_for(:artigo) }
end
```

O `attributes_for` nos retorna um hash com o valor definido em nossa factory e, assim como o `create`, podemos alterar os valores ao invocá-lo, passando um hash opcional.

```
attributes_for(:artigo, titulo: 'Novo título')
```

Utilizando o `attributes_for`, nosso teste de controller fica bem mais enxuto.

```
describe "POST 'create'" do
  let(:params) do
    { artigo: attributes_for(:artigo) }
  end

  it 'create a new artigo' do
    expect do
      artigo :create, params
    end.to change(Artigo, :count).by(1)
  end
end
```

3.6 Sendo DRY

Herança

Até o momento, criamos apenas uma factory para cada model, mas é comum definirmos mais factories para facilitar o nosso uso e não precisar ficar passando sempre os mesmos parâmetros.

Vamos agora definir duas factories para `Artigo`. Uma de um artigo aprovado e outra para um artigo não aprovado. Ao utilizarmos um bloco `factory` dentro de outro, definimos uma factory que herda os valores da factory pai.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo 'Contenteúdo de Diversas dicas do RSpec'

  factory :artigo_aprovado do
    aprovado true
  end

  factory :artigo_nao_aprovado do
    aprovado false
  end
end
```

Ao chamarmos `FactoryGirl.create(:artigo_aprovado)`, esta terá todos os valores definidos na factory `artigo` mais os definidos em `artigo_aprovado`.

Veja um exemplo ao utilizarmos `FactoryGirl.create(:artigo_aprovado)`.

```
#<Artigo:0x000001090c8668> {
            :id => 9,
         :titulo => "Diversas dicas do RSpec",
       :conteudo => "Contenteúdo de Diversas dicas do RSpec",
    :created_at => Thu, 16 Jan 2014 21:05:18 UTC +00:00,
    :updated_at => Thu, 16 Jan 2014 21:05:18 UTC +00:00,
      :aprovado => true
}
```

Se usarmos apenas `FactoryGirl.create(:artigo)`, o valor de `aprovado` será `nil`.

Este é um bom modo de mantermos a boa prática de definir apenas os valores obrigatórios quando declaramos a factory base, e nas factories definidas via herança passamos os valores não obrigatórios.

Sendo melhor, utilizando traits

Herança é muito legal! No entanto, imagine que tenhamos uma factory com um título todo em caixa alta que seja aprovada e uma outra factory não aprovada. Além disso, temos que manter a versão que já funciona com o titulo do modo que está.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo 'Contenteúdo de Diversas dicas do RSpec'

  factory :artigo_aprovado do
    aprovado true
  end

  factory :artigo_nao_aprovado do
    aprovado false
  end

  factory :artigo_aprovado_titulo_maiusculo do
    titulo 'DIVERSAS DICAS DO RSPEC'
    aprovado true
  end

  factory :artigo_nao_aprovado_titulo_maiusculo do
    titulo 'DIVERSAS DICAS DO RSPEC'
    aprovado false
  end
end
```

Como se pode ver, começamos a criar muita repetição, o campo `titulo` e `aprovado` foram duplicados com o mesmo valor.

Vamos refatorar agora para usarmos traits. Traits, assim como ao declararmos a factory, também recebe um bloco. Desta forma, vamos remover as declarações das outras factories: deixamos apenas a factory `artigo` e criamos uma trait para cada uma das nossas necessidades.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
```

```
  conteudo 'Contenteúdo de Diversas dicas do RSpec'

  trait :aprovado do
    aprovado true
  end

  trait :nao_aprovado do
    aprovado false
  end

  trait :titulo_maiusculo do
    titulo 'DIVERSAS DICAS DO RSPEC'
  end
end
```

Vamos agora ver como utilizar a trait para criarmos a nossa factory antiga `artigo_aprovado_titulo_maiusculo`. Para isso, passamos apenas os símbolos com os nomes das traits.

```
FactoryGirl.create(:artigo, :aprovado, :titulo_maiusculo)
```

Vale lembrar que ainda podemos passar um hash opcional se quisermos alterar qualquer um dos parâmetros.

```
FactoryGirl.create(:artigo, :aprovado, :titulo_maiusculo,
    conteudo: 'Novo conteúdo')
```

No nosso exemplo, passamos apenas um parâmetro, no entanto, é possível passar quantos forem necessários.

Agora não temos mais repetição de nenhum campo. Se não gostar muito de ter que sempre ficar passando a trait, e preferir chamar `artigo_aprovado_titulo_maiusculo`, é possível fazer isto, mas desta vez sem repetição.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo 'Contenteúdo de Diversas dicas do RSpec'

  trait :aprovado do
```

```
    aprovado true
  end

  trait :nao_aprovado do
    aprovado false
  end

  trait :titulo_maiusculo do
    titulo 'TITLE'
  end

  factory :artigo_aprovado_titulo_maiusculo,
    traits: [:aprovado, :titulo_maiusculo]
end
```

E utilizamos a factory como fizemos anteriormente, `FactoryGirl.create(:artigo_aprovado_titulo_maiusculo)`.

3.7 Atributos dinâmicos nas factories

Até o momento, definimos em nossas factories os atributos manualmente, mas seria legal se pudéssemos brincar um pouco utilizando atributos mais dinâmicos, afinal estamos programando.

Lazy Attributes

Podemos passar um bloco para os atributos das factories quando queremos que estes valores sejam avaliados a cada vez que uma instância é criada.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo 'Contenteúdo de Diversas dicas do RSpec'
  created_at { 2.days.ago }
end
```

Alteramos o valor de `created_at` para chamar `2.days.ago`, de modo que, sempre que um registro for criado, terá o valor de `created_at` a data de 2 dias atrás.

Dependent Attributes

Vamos alterar nossa factory de artigo para definirmos o valor de `conteudo` dinamicamente, exibindo informações do `titulo` e do `aprovado`. Para isso, passamos um bloco, afinal será um atributo lazy. Dentro deste bloco temos acesso aos outros atributos do model.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo
    { "Conteudo do artigo #{titulo}. Approved: #{aprovado}" }
end
```

Observe que alteramos o nosso `conteudo` para ter um valor mais relevante, exibindo o `titulo` e o valor de `aprovado`. Pode parecer meio desnecessário declarar `conteudo` com um valor mais descritivo, mas todo o valor de se ter um `conteudo` dinâmico é visto quando estamos debugando, afinal, temos mais informações úteis do objeto de uma maneira fácil de ser lida.

Sequences

Até o momento, o nosso campo `titulo` é o mesmo para todas as factories. Vamos alterar para este valor também ser dinâmico, mas agora não baseado em outros campos e, sim, em uma sequência de valores.

Primeiro definimos um `sequence`, passando um symbol com o nome do campo que queremos utilizar, e passamos um bloco para gerar este conteúdo do campo `titulo`. Na definição de nossa factory, simplesmente usamos `titulo`, que utiliza o valor gerado pela sequence.

```
FactoryGirl.define do

  sequence :titulo do |n|
    "Diversas dicas do RSpec #{n}"
  end

  factory :artigo do
    titulo
```

```
    conteudo
      { "Conteudo do artigo #{titulo}. Approved: #{aprovado}" }
  end
end
```

Deste modo, nossas factories geradas terão os títulos: "Diversas dicas do RSpec 1", "Diversas dicas do RSpec 2", "Diversas dicas do RSpec 3" etc.

In line sequence

Como estamos utilizando apenas o valor de `sequence` para a factory de artigo, podemos definir este valor inline, com o `sequence` diretamente dentro do bloco de `factory`.

```
factory :artigo do
  sequence(:titulo) { |n| "Diversas dicas do RSpec #{n}" }
  conteudo
    { "Conteudo do artigo #{titulo}. Approved: #{aprovado}" }
end
```

Deste modo, economizamos algumas linhas e fica mais claro o que está acontecendo.

Podemos alterar o valor inicial simplesmente passando o primeiro valor para o `sequence`, da seguinte maneira:

```
factory :artigo do
  sequence(:titulo, 125) { |n| "Diversas dicas do RSpec #{n}" }
  conteudo
    { "Conteudo do artigo #{titulo}. Approved: #{aprovado}" }
end
```

Assim, nossas factories geradas terão os títulos: "Diversas dicas do RSpec 125", "Diversas dicas do RSpec 126", "Diversas dicas do RSpec 127" etc.

É possível passar qualquer objeto que implemente o método `#next` assim podemos passar uma string.

```
factory :artigo do
  sequence(:titulo, 'a') { |n| "Diversas dicas do RSpec #{n}" }
  conteudo
```

```
    { "Conteudo do artigo #{titulo}. Approved: #{aprovado}" }
end
```

Como você deve imaginar, os nossos títulos serão gerados como: "Diversas dicas do RSpec a", "Diversas dicas do RSpec b", "Diversas dicas do RSpec c" etc.

3.8 Associações

No mundo Rails, é bem comum termos que usar as associações do Active Record, para ligarmos uma entidade à outra. Vamos ver como podemos criar estas associações utilizando as factories. Primeiro, nossa factory de `Usuario`.

```
factory :usuario do
  nome 'Mauro'
  email { "#{nome}@helabs.com.br" }
end
```

Agora, nossa factory de `Artigo`. O nosso relacionamento é de um `Artigo` que pertence a (`belongs_to`) um `Usuario` e um `Usuario` tem vários (`has_many`) `Artigos`.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo { "Conteudo do #{titulo}" }

  trait :aprovado do
    aprovado true
  end

  factory :artigo_aprovado, traits: [:aprovado]
end
```

Criando a associação diretamente

Podemos criar a associação no momento em que estamos utilizando a nossa factory, da seguinte maneira:

```
usuario = FactoryGirl.create(:usuario)
FactoryGirl.create(:artigo, usuario: usuario)
```

Primeiro, criamos uma instância de `Usuario` e passamos para factory de `Artigo`, no atributo `usuario`, um objeto `Usuario`. Dessa forma, a associação é criada automaticamente e conseguimos criar um usuário que possui um artigo criado.

Definindo a associação

Dado que todo `Artigo` é escrito por um usuário, não seria legal termos em nossa factory um `Artigo` sem usuário, se não, sempre que fôssemos utilizar a factory de `Artigo`, teríamos que ficar criando um usuário e passar explicitamente como fizemos no exemplo anterior. Seria melhor se definíssemos isso na nossa factory diretamente. É o que fazemos simplesmente declarando abaixo de `conteudo` o `usuario`. Não passamos nenhum valor para `usuario`, mas como existe uma factory com este nome, a factory_girl cria a associação automaticamente.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo { "Conteudo do #{titulo}" }
  usuario

  trait :aprovado do
    aprovado true
  end

  factory :artigo_aprovado, traits: [:aprovado]
end
```

Com isso, a factory_girl irá criar sempre um usuário associado a um artigo, quando utilizarmos a factory de artigo.

Definindo a factory na associação

Nossa associação entre `Usuario` e `Artigo` ainda não está legal, afinal não temos o usuário do artigo mais sim o autor do artigo. Vamos alterar nosso model de forma que reflita isso.

```
class Artigo < ActiveRecord::Base
```

```
  belongs_to :autor, class_name: 'Usuario'
end
```

Vale lembrar que para isso funcionar devemos ter a coluna `autor_id` na classe `Artigo`, por isso renomeamos a nossa coluna `usuario_id` para `autor_id`.

Agora que temos a associação de que `Artigo` pertence a um `Usuario`, se ela for nomeada `autor` nossa factory quebrará.

Vamos definir a associação na nossa factory trocando a chamada de `usuario` para `association`, passando o nome da associação e qual factory queremos definir para a tal.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo { "Conteudo do #{titulo}" }
  association :autor, factory: :usuario

  trait :aprovado do
    aprovado true
  end

  factory :artigo_aprovado, traits: [:aprovado]
end
```

Podemos alterar a factory de usuário passando os parâmetros adicionais da seguinte maneira:

```
association :autor, factory: :usuarios, nome: 'Mauro George'
```

Aliases

No exemplo anterior, utilizamos o método `association` para definir a factory da associação. Não podemos nos referenciar a `autor` diretamente, dado que nossa factory está declarada como `usuario`.

Podemos declarar um alias para nossa factory de `Usuario` ,assim poderemos utilizar `autor` diretamente. Para isso, simplesmente passamos um array para a chave `aliases`.

```
factory :usuario, aliases: [:autor] do
  nome 'Mauro'
  email { "#{nome}@helabs.com.br" }
end
```

Dessa forma, além de podermos utilizar a factory como `FactoryGirl.create(:usuario)`, é possível usar `FactoryGirl.create(:autor)`.

Podemos alterar a nossa factory de artigo para utilizar a factory de autor diretamente, que nada mais é do que um alias da factory de usuário.

```
factory :artigo do
  titulo 'Diversas dicas do RSpec'
  conteudo { "Conteudo do #{titulo}" }
  autor

  trait :aprovado do
    aprovado true
  end

  factory :artigo_aprovado, traits: [:aprovado]
end
```

Definindo uma associação has_many

No exemplo anterior, definimos na factory uma associação `belongs_to`, onde dizemos que um `Artigo` pertence a `Usuario`. Vamos agora definir uma associação `has_many`, para um `Usuario` que tem vários `Artigos`.

Nossa primeira tentativa provavelmente seria definir o `artigo` diretamente.

```
factory :usuario, aliases: [:autor] do
  nome 'Mauro'
  email { "#{nome}@helabs.com.br" }
  artigo
end
```

Ao tentarmos rodar o código usando `FactoryGirl.create(:autor)`, recebemos o seguinte erro do Active Record: `NoMethodError: undefined method 'artigo='for #<Usuario:0x000001090012e8>'`. Afinal, o nosso model `Usuario` não implementa `artigo=`, e sim `artigos=`, dado que temos `has_many :artigos`.

Não adianta tentarmos trocar `artigo` para `artigos`, pois não obteremos sucesso. Isso porque a factory_girl tentará encontrar uma trait com o nome de artigos em vez de criar a associação.

Conseguimos criar a associação diretamente como fizemos no exemplo anterior, utilizando:

```
autor = FactoryGirl.create(:autor)
FactoryGirl.create(:artigo, autor: autor)
```

Mas como definimos uma associação has_many diretamente na factory?

Conhecendo os callbacks

Assim como o Active Record, a factory_girl também possui os callbacks. Utilizaremos o callback `after(:create)` para resolver o nosso problema e conseguir criar uma associação `has_many` na definição de nossa factory.

Definimos primeiro uma trait, apenas porque não queremos que todos os usuários tenham um artigo criado, já que não é um dado obrigatório. Lembra da boa prática né?

Na nossa trait, chamamos o callback `after(:create)`, que recebe um bloco, e como primeiro argumento ele nos passa a instância da factory criada. Então, simplesmente passamos o nosso `Usuario` para a criação de um artigo. Do mesmo modo que fizemos ao utilizar a factory diretamente, agora conseguimos fazer isso na definição da factory.

```
factory :usuario, aliases: [:autor] do
  nome 'Mauro'
  email { "#{nome}@helabs.com.br" }

  trait :com_artigo do
    after(:create) do |usuario|
```

```
      create(:artigo, autor: usuario)
    end
  end
end
```

O problema é que todo autor terá apenas um artigo criado. Seria bom se conseguíssemos aumentar este número.

O create_list

Se precisássemos criar uma coleção de artigos, provavelmente utilizaríamos `#times`.

```
3.times { FactoryGirl.create(:artigo) }
```

No entanto, a factory_girl nos dá o método `create_list`. Assim, podemos trocar o uso do `times` para:

```
FactoryGirl.create_list(:artigo, 3)
```

Ficou fácil alterar nossa factory para criar cada autor com pelo menos 3 artigos.

```
factory :usuario, aliases: [:autor] do
  nome 'Mauro'
  email { "#{nome}@helabs.com.br" }

  trait :com_artigo do
    after(:create) do |usuario|
      create_list(:artigo, 3, autor: usuario)
    end
  end
end
```

Agora sim, todos autores têm 3 artigos. É... mas e se quisermos um autor com 4 artigos? #comofas ?

Olá Transient Attributes

A factory_girl nos dá a opção de definirmos atributos que não estão em nosso model. Para isso, utilizamos um bloco no método `ignore` e, se utilizarmos estes atributos em conjunto com o callback, conseguiremos definir a quantidade de artigos que queremos criar dinamicamente.

Para isso, passamos para o bloco `ignore` o nosso *transient attribute*, o `total_de_artigos` e definimos o valor 3. Passamos para o nosso bloco do `after(:create)` um novo parâmetro, o `evaluator`, que armazena todos os valores da factory, inclusive os ignorados. Assim, passamos o valor de `total_de_artigos` para o `create_list`.

```
factory :usuario, aliases: [:autor] do
  nome 'Mauro'
  email { "#{nome}@helabs.com.br" }

  trait :com_artigo do
    ignore do
      total_de_artigos 3
    end

    after(:create) do |usuario, evaluator|
      create_list(:artigo, evaluator.total_de_artigos,
                  autor: usuario)
    end
  end
end
```

Podemos agora criar usuários com a quantidade de artigos que desejarmos da seguinte maneira:

```
FactoryGirl.create(:usuarios, :com_artigo, total_de_artigos: 10)
```

Se omitirmos o parâmetro `total_de_artigos`, ele será criado com o valor padrão, que definimos como 3.

```
FactoryGirl.create(:usuarios, :com_artigo)
```

3.9 Bah, mas só funciona com Active Record?

Curtiu bastante a factory_girl e agora quer usar factory para gerar conteúdo que normalmente é repetido, né? Vamos fazer isso então.

Quando trabalhamos com a http://pokeapi.co, vimos que ela retorna um JSON. Diversos colaboradores do nosso sistema estão acessando este JSON, então será muito comum vermos em diversos testes algo como o seguinte para declarar a mesma string, ou alterar alguns campos em diversos lugares:

```
json = %({"national_id": 6, "name": "Charizard", "attack":84,
          "defense":78})
```

Por padrão, a factory_girl é para definição de objetos Active Record. Vamos ver como fazemos para a factory_girl trabalhar com outros objetos.

Vamos primeiro definir as propriedades de que precisamos:

```
factory :pokeapi do
  id_nacional 6
  nome 'Charizard'
  ataque 84
  defesa 78
end
```

Ao tentarmos executar esta factory diretamente, receberemos `NameError: uninitialized constant Pokeapi`, porque a factory_girl está tentando criar uma instância de `Pokeapi` e, como não existe este model, o erro é lançado.

Nosso JSON nada mais é do que uma string ruby, então vamos dizer isto à factory_girl para ver se ajuda.

```
factory :pokeapi, class: String do
  id_nacional 6
  nome 'Charizard'
  ataque 84
  defesa 78
end
```

Novamente, ao executarmos a factory_girl, recebemos erro, mas agora o erro é `NoMethodError: undefined method 'id_nacional='for`

`"":String`. Isso porque a factory_girl está tentando passar cada um dos atributos à classe como faz com o Active Record, mas não é assim que instanciamos uma string! Vamos alterar novamente nossa factory,, agora para ter um `initializer` customizado. Para isso utilizaremos o método `initialize_with`, que recebe um bloco, dentro do qual instanciamos a nossa classe String. Para criar a nossa classe, primeiro criamos um hash com cada um dos atributos definidos na factory e, em seguida, passamos este hash para o `JSON.generate`, que transforma o hash em um JSON retornando uma string. Adicionamos todos os nossos atributos no bloco `ignore`, pois usaremos os nossos atributos no `initialize_with`, e não na string diretamente.

```
factory :pokeapi, class: String do
  ignore do
    # ...
  end

  initialize_with do
    info = { national_id: id_nacional, name: nome,
             attack: ataque, defense: defesa }
    JSON.generate(info)
  end
end
```

Vamos agora utilizar nossa factory e... recebemos outro erro! Caramba! Estou achando que isso não vai funcionar =/

Agora o erro foi `NoMethodError: undefined method 'save!'for #<String:0x00000108b28eb0>`, ou seja, a factory_girl ainda pensa que estamos usando o comportamento de um objeto Active Record e tenta chamar `save!` em nossa string.

Para resolvermos isso, simplesmente utilizamos o método `skip_create`, que define que nossa factory não chamará o `save!`.

```
factory :pokeapi, class: String do
  skip_create

  ignore do
```

```
    # ...
  end

  # ...
end
```

Agora sim nossa factory está funcionando! E retornando o nosso JSON. Utilizamos a factory como qualquer outra.

```
FactoryGirl.create(:pokeapi)
```

Podemos passar qualquer um dos parâmetros como fizemos anteriormente.

```
FactoryGirl.create(:pokeapi, nome: 'Bulbasaur')
```

Agora que nossa factory está funcionando podemos fazer tudo de legal que já vimos utilizando a factory_girl como traits.

3.10 Conhecendo as estratégias

Até o momento, utilizamos duas estratégias, mesmo sem saber que esse era o nome delas. Foram o `create` e o `attributes_for`. O `create` salva o objeto no banco de dados e o `attributes_for` nos retorna um hash como vimos anteriormente.

É comum utilizarmos sempre o `create`, mas realmente sempre precisamos dos dados persistidos no banco de dados? Vamos criar um método `#nome_completo` para o nosso model `Pokemon`. Primeiro, vamos ao teste. Verificamos que o método `#nome_completo` é o nome do pokémon seguido do seu `nacional_id` separado por um traço.

```
describe '#nome_completo' do

  let(:pokemon) do
    create(:pokemon)
  end

  subject do
```

```
    pokemon.nome_completo
  end

  it 'exibe o nome e o id nacional' do
    expect(subject).to eq('Charizard - 6')
  end
end
```

Vamos agora fazer este teste passar, criando o método `#nome_completo`, que simplesmente concatena o `nome` e o `id_nacional`.

```
class Pokemon < ActiveRecord::Base

  def nome_completo
    "#{nome} - #{id_nacional}"
  end
end
```

Nosso teste passa, no entanto, estamos criando um registro no banco de dados toda vez que o teste é rodado, e isso é lento. Vamos ver como podemos melhorar.

Conhecendo o build

Uma das estratégias da factory_girl é o `build`. Diferentemente do `create`, o build apenas cria o objeto Active Record não o persistindo no banco de dados. Vamos ver um exemplo.

```
pokemon = FactoryGirl.build(:pokemon)
```

Podemos ver se nosso objeto está salvo ou não simplesmente usando o método `persisted?` do Active Record.

```
pokemon.persisted?
```

No nosso caso, é retornado `false`. Podemos pensar que o `build` da factory_girl seria o mesmo que utilizarmos o `new` do Active Record, passando os atributos definidos na factory.

Podemos refatorar nosso teste para usar o `build` simplesmente alterando o nosso `let`.

```
let(:pokemon) do
  build(:pokemon)
end
```

Nossos testes continuam passando, dado que eles não têm dependência nenhuma com o banco de dados, apenas com os atributos definidos em nosso objeto.

build_stubbed, o irmão mais poderoso do build

Uma outra estratégia disponibilizada pela factory_girl é o `build_stubbed`. Diferente do `build`, não estamos criando um objeto Active Record real, mas sim fazendo stub de seus métodos. Em consequência, esta estratégia é a mais rápida de todas. Vamos a um exemplo.

```
pokemon = FactoryGirl.build_stubbed(:pokemon)
```

Diferentemente do `build`, nosso objeto age como estivesse persistido, por isso, ao usarmos `pokemon.persisted?`, nosso resultado será `true`. Nosso objeto somente age como se estivesse persistido, pois se fizermos um `Pokemon.count` antes e depois do uso do `build_stubbed` obteremos o mesmo valor, mesmo que explicitamente salvemos nosso objeto com `#save!`. Com o `build`, realmente o objeto é salvo no banco de dados se usarmos `#save!`.

Como nosso teste não depende diretamente do banco de dados, podemos alterá-lo para utilizar o `build_stubbed`.

```
let(:pokemon) do
  build_stubbed(:pokemon)
end
```

O código normalmente deve depender menos do seu estado em relação ao banco de dados e depender mais do seu estado em relação a outros objetos. Sendo assim a não ser que estejamos testando métodos que façam consultas no banco de dados, é uma boa pedida utilizar o `build_stubbed`.

3.11 E quando as factories não são mais válidas?

Vimos no início do capítulo que é uma boa prática definirmos apenas os atributos que são obrigatórios devido à validação do Active Record. Mas é comum, no nosso desenvolvimento, termos que adicionar novos campos devido a novas validações do nosso model.

Vamos supor que agora o nosso model `Pokemon` ganhou um novo campo `ataque` e este é obrigatório. Vamos à validação.

```
class Pokemon < ActiveRecord::Base

  validates :ataque, presence: true
end
```

Agora ao tentarmos rodar nossa factory de Pokémon receberemos o seguinte erro do Active Record:

```
ActiveRecord::RecordInvalid:
  Validation failed: Attack can't be blank
```

Imagine agora isso na sua suíte de testes e uma boa base delas utilizando a factory de pokémon. Teremos diversos testes quebrados.

Para evitar isso, criamos testes para as nossas factories da seguinte maneira: primeiro criamos um arquivo em `spec/factories_spec.rb` e utilizamos o método `FactoryGirl.factories` para obter todas as factories declaradas. Em seguida, iteramos sobre cada uma destas factories criando um teste para cada uma, dinamicamente. E verificamos se a factory responde a `valid?` antes de fazermos a asserção, pois nem todas as factories herdam de `ActiveRecord::Base`, como vimos com a factory `pokeapi`.

```
describe 'FactoryGirl' do
  FactoryGirl.factories.map(&:name).each do |factory_nome|
    it "factory #{factory_nome} is valid" do
      factory = build(factory_nome)

      if factory.respond_to?(:valid?)
        expect(factory).to be_valid
      end
```

```
    end
  end
end
```

Agora temos testes para as factories, no entanto, nossa suíte roda as specs em ordem aleatória. Sendo assim, nossas specs das factories ficaram perdidas no meio de todas as outras specs e continuaremos com um monte de specs quebradas. Podemos rodar apenas a spec das factories para ver se o problema é lá, mas é possível automatizar isso, rodando as specs das factories antes da nossa suíte. Para tanto, adicionamos o seguinte código a nosso `Rakefile`:

```
if defined?(RSpec)
  desc 'Run factory specs.'
  RSpec::Core::RakeTask.new(:factory_specs) do |t|
    t.pattern = './spec/factories_spec.rb'
  end
end

task spec: :factory_specs
```

Que simplesmente roda as specs das factories antes de rodar as demais specs e, caso alguma spec de factory quebre, ele nem inicia os demais testes. É exatamente o que precisamos: assim, se uma de nossas factories quebrar, não ficaremos assustados ao ver uma boa parte de nossa suíte de teste quebrando.

Conhecendo o lint

Aprendemos anteriormente como testar nossas factories. No entanto, é chato ter que ficar copiando isto para todos os projetos em que iremos trabalhar. E se automatizarmos isto? Afinal, é um padrão. Foi pensando nisso que o pessoal que trabalha no factory_girl criou o `FactoryGirl.lint`. Ele faz exatamente o que fizemos manualmente: roda nossas specs de factories antes das specs do app.

Para utilizar o lint, adicionamos um bloco `before(:suite)` em nosso `spec_helper.rb`, chamando o `FactoryGirl.lint`.

```
RSpec.configure do |config|
  # ...
```

```
  config.before(:suite) do
    FactoryGirl.lint
  end
end
```

Agora temos o mesmo resultado, no entanto, de uma maneira mais enxuta e portável entre diversos projetos.

3.12 Conclusão

Espero que tenha gostado da factory_girl e de todas as técnicas que vimos aqui. Com certeza isso ajudará muito quando estiver escrevendo os seus testes. Neste capítulo:

- Vimos as vantagens e desvantagens do uso de factory sobre fixtures;
- Definimos e utilizamos nossa primeira factory;
- Utilizamos nossa factory nos testes;
- Configuramos o factory_girl para ser menos verboso durante os testes;
- Utilizamos herança e trait para não nos repetirmos ao declararmos as nossas factories;
- Utilizamos atributos dinâmicos nas factories;
- Criamos associações básicas entre factories;
- Definimos uma associação complexa utilizando callback, trait, create_list e transient attributes;
- Utilizamos a factory_girl para gerar um objeto que não é Active Record;
- Conhecemos as diversas estratégias que a factory_girl possui;
- Aprendemos testar as nossas factories para evitarmos surpresas desagradáveis.

Continue por aí que no próximo capítulo viajaremos no tempo utilizando o timecop. Veremos por que devemos tomar cuidado com testes que dependam de data e como podemos utilizar o timecop para nos ajudar.

Capítulo 4

Precisamos ir... de volta para o futuro

4.1 Introdução

Nosso app de batalhas pokémons está evoluindo. Temos que implementar uma nova funcionalidade que é selecionar todos os pokémons que foram selecionados no dia anterior. Utilizaremos estes dados para exibir em algum momento no sistema.

Vamos primeiro ao nosso teste. Criaremos um método `Pokemon.escolhidos_ontem`. Primeiro, definimos o nosso cenário, que consiste de 3 pokémons: um que foi escolhido hoje, um que foi escolhido ontem e um que foi escolhido antes de ontem. Para isso, definiremos o valor de `Pokemon#escolhido_em` utilizando `Time.zone.local`.

```
describe '.escolhidos_ontem' do

    let!(:pokemon_escolhido_hoje) do
      create(:pokemon,
          escolhido_em: Time.zone.local(2015, 1, 8, 1))
    end

    let!(:pokemon_escolhido_ontem) do
      create(:pokemon,
        escolhido_em: Time.zone.local(2015, 1, 2, 23, 59, 59))
    end

    let!(:pokemon_escolhido_antes_de_ontem) do
      create(:pokemon,
          escolhido_em: Time.zone.local(2014, 1, 2))
    end
end
```

Tendo definido o nosso cenário, vamos definir agora o sujeito do nosso teste e o primeiro teste, de que o `Pokemon.escolhidos_ontem` deve retornar os pokémons que foram escolhidos no dia anterior.

```
subject do
  Pokemon.escolhidos_ontem
end

it 'tem o pokemon escolhido ontem' do
  expect(subject).to include(pokemon_escolhido_ontem)
end
```

Rodamos o nosso teste e ele quebra, dado que ainda não definimos o nosso método. Vamos criá-lo agora.

Como necessitamos de um método de classe que retorne uma coleção de pokémons, podemos utilizar o `scope` do Active Record. Criamos um escopo chamado `escolhidos_ontem` que faz um `where` retornando os pokémons criados entre meia-noite do dia anterior e meia-noite de hoje, ou seja, ontem.

```
class Pokemon < ActiveRecord::Base
```

```
  scope :escolhidos_ontem, -> do
    where(escolhido_em:
          1.day.ago.midnight..Time.zone.now.midnight)
  end
end
```

Rodamos novamente nosso teste e agora ele passa. Vamos implementar agora os testes dos pokémons que ele não deve incluir, que são os que foram escolhidos hoje e os que foram escolhidos antes de ontem.

```
it 'não tem o pokemon hoje' do
  expect(subject).to_not include(pokemon_escolhido_hoje)
end

it 'não tem o pokemon escolhido antes de ontem' do
  expect(subject).
  to_not include(pokemon_escolhido_antes_de_ontem)
end
```

Rodamos nossa suíte novamente e todos os testes continuam verdes. Terminamos nosso trabalho por hoje, no entanto, na manhã do dia seguinte ao rodar os testes novamente recebemos erros nos testes que antes passavam! Mas por quê? Se observarmos, nos nossos testes definimos o dia de hoje como 1/02/2015, e o utilizamos no `pokemon_escolhido_hoje`. Durante todo aquele dia nossa suíte de testes funcionará mas, em qualquer outro dia, ela quebrará, dado que não estaremos mas no dia 2/02/2015.

4.2 Congelando o tempo com timecop

A gem timecop nos dá exatamente os poderes de que necessitamos, de parar o tempo em um momento especifico. Se conseguíssemos manter o tempo sempre no dia 2/02/2015, nossos testes sempre passariam, dado que naquele dia eles passavam.

Instalação

Vamos instalar o timecop simplesmente adicionando-o ao nosso Gemfile:

```
gem 'timecop'
```

E finalizamos rodando `bundle`.

Congelando o tempo

Agora que temos o timecop instalado, vamos aos nossos testes. O timecop nos fornece o método `Timecop.freeze` para pararmos o tempo. Passamos como primeiro parâmetro o lugar do tempo que queremos parar e um bloco com o contexto do que deve ser parado no tempo.

```
it 'tem o pokemon escolhido ontem' do
  Timecop.freeze(Time.zone.local(2015, 1, 2, 12)) do
    expect(subject).to include(pokemon_escolhido_ontem)
  end
end
```

Agora todo código que é executado dentro do bloco do `Timecop.freeze` é interpretado como se a data fosse 7/03/2014 às 12:00:00. Fizemos no primeiro teste, mas temos que aplicar o timecop aos demais testes também.

```
it 'não tem o pokemon hoje' do
  Timecop.freeze(Time.zone.local(2015, 2, 2, 12)) do
    expect(subject).to_not include(pokemon_escolhido_hoje)
  end
end

it 'não tem o pokemon escolhido antes de ontem' do
  Timecop.freeze(Time.zone.local(2015, 3, 3, 12)) do
    expect(subject).
      to_not include(pokemon_escolhido_antes_de_ontem)
  end
end
```

Observe que estamos repetindo diversas vezes a definição de hoje com o `Time.zone.local(2015, 3, 3, 12)` e está sendo um valor mágico, afinal, não sabemos o que esta data quer dizer a não ser que analisemos o contexto todo, lendo os demais testes.

Podemos melhorar isso definindo o `hoje`, utilizando o `let`.

```
let(:hoje) do
  Time.zone.local(2014, 3, 3, 12)
end
```

Em seguida, alteramos todos os nossos testes que usam o timecop para utilizarem o valor definido n o `let`.

```
it 'tem o pokemon escolhido ontem' do
  Timecop.freeze(hoje) do
    expect(subject).to include(pokemon_escolhido_ontem)
  end
end
```

Dado que o timecop nos dá o poder de parar o tempo, é uma boa prática definirmos o nosso "hoje" sempre no passado, pois assim evitamos qualquer surpresa de os testes passarem no dia e começarem a falhar depois de um tempo. Para facilitar nosso trabalho, alteramos apenas o ano de 2014 para 2010 do nosso `hoje`.

```
let(:hoje) do
  Time.zone.local(2000, 3, 3, 12)
end
```

Por que tantos lets?

Vamos olhar para nosso teste. Perceba que temos definido todo o nosso cenário dos testes em diversos `let` antes de qualquer um deles. No entanto, vamos analisar o nosso primeiro teste.

```
it 'tem o pokemon escolhido ontem' do
  Timecop.freeze(hoje) do
    expect(subject).to include(pokemon_escolhido_ontem)
  end
end
```

Perceba que em nenhum momento estamos utilizando o `pokemon_escolhido_ontem` e o `pokemon_escolhido_antes_de_ontem`, mas apenas o `pokemon_escolhido_ontem`. Então, por que declaramos todos eles

antes? Normalmente por não fazermos TDD, pois, no nosso exemplo, criamos todos os cenários para só depois iniciarmos os testes.

Aparentemente não faz mal nenhum, correto? Errado! Lembra quando falamos que factories são lentas 3.1? Fazendo este tipo de abordagem ajuda a tornar um teste lento, afinal estamos criando objetos no banco de dados que não são nem usados no teste. Isso também deixa os testes frágeis, pois estamos criando um cenário grande que é difícil de se manter em testes que testam ordenação, por exemplo.

Vamos alterá-lo para definir apenas os dados que são usados por eles. Para isso, removemos o `let` e criamos o nosso objeto apenas dentro do bloco do teste, que é onde ele é utilizado somente.

```
it 'tem o pokemon escolhido ontem' do
  pokemon_escolhido_ontem = create(:pokemon,
      escolhido_em: Time.zone.local(2010, 3, 3, 23, 59, 59))
  Timecop.freeze(hoje) do
    expect(subject).to include(pokemon_escolhido_ontem)
  end
end
```

Agora nossos objetos são criados apenas quando necessários. Por isso, é sempre bom fazermos o TDD, e se percebermos uma repetição, extraímos para um `let`, como foi o caso do `hoje`. Em muitos dos casos é possível utilizar o `let` para definir o cenário, apenas deixe isso surgir. Quando perceber que muito dos seus testes estão declarando o mesmo objeto, mova para um `let` mas não inicie gerando todo o cenário.

4.3 Removendo repetição

Olhando para nossos testes, vemos sempre a chamada a `Timecop.freeze` em todos eles. O timecop nos dá a possibilidade de utilizarmos uma sintaxe sem a necessidade de um bloco. Para isso, vamos definir um bloco `before`, dado que todo o nosso contexto depende da data, com a chamada a `Timecop.freeze`, e um bloco `after`, em que utilizaremos o `Timecop.return` que volta o tempo para o nosso hoje real.

```
before do
  hoje = Time.zone.local(2010, 3, 3, 12)
  Timecop.freeze(hoje)
end

after do
  Timecop.return
end
```

Perceba que movemos a definição d o `hoje` para dentro d o `before`, já que não iremos utilizá-lo em mais de um lugar. Para finalizar, removemos o `Timecop.freeze` dos nossos testes.

```
it 'tem o pokemon escolhido ontem' do
  pokemon_escolhido_ontem = create(:pokemon,
      escolhido_em: Time.zone.local(2010, 2, 3, 23, 59, 59))
  expect(subject).to include(pokemon_escolhido_ontem)
end
```

Agora removemos a repetição dos testes. No entanto, temos que lembrar que sempre que utilizamos o timecop sem o bloco temos que usar o `Timecop.return` pois se não, o tempo não é retornado para o hoje atual. Como os testes são rodados em ordem aleatória, o tempo continuará o definido no timecop mesmo em testes que não o estão utilizando, o que pode gerar testes quebradiços que quebram aleatoriamente. O pior é que é muito difícil de se identificar o problema, visto que nenhuma exceção é lançada. Com alguns *seeds* os testes passarão e com outros não.

```
describe '.outro_metodo' do

  it 'faz alguma coisa' do
    puts Time.zone.now
  end
end
```

Nesse exemplo, removemos o bloco `after`, para testes apenas. Em algum momento, o valor de `Time.zone.now` pode ser `2000-03-03 12:00:00 -0300`, o valor que foi definido pelo timecop em outro contexto de testes, ou

o valor real. Sendo assim, cabe a nós desenvolvedores ficarmos atentos ao uso de cada uma de nossas ferramentas para não gerarmos problemas para todo o time por um simples descuido.

Ordem aleatória nos testes

Uma boa prática é sempre rodarmos os testes em ordem aleatória. Isso porque testes que estejam dependendo de outros são um problema, pois não sabemos onde está definida toda sua estrutura, de modo que não fica claro o que está sendo testado, e nem é prático de se alterar algo, dado que uma alteração pode fazer outros testes quebrarem.

Por padrão, ao utilizarmos o gerador do rspec-rails, ele adiciona a seguinte linha no `spec_helper.rb`:

```
config.order = "random"
```

Ela define que nossos testes serão executados em ordem aleatória. Sendo assim, é uma boa verificar em projetos mais antigos se eles possuem esta linha e adicioná-la caso ainda não tiver.

O RSpec nos permite definir suas configurações utilizando um arquivo `.rspec` na raiz do projeto, ou no nosso diretório *home*. Podemos definir no nosso home para sempre rodar os testes em ordem aleatória, o que nos ajuda a encontrar problemas, já que podemos esquecer de fazer a verificação em projetos antigos. Neste arquivo simplesmente adicionamos:

```
--order random
```

Se encontrarmos qualquer problema ao rodarmos os testes, podemos fazer uso do `seed` gerado para debug utilizando `rspec spec --seed 182`, simplesmente trocando o 182 pelo valor gerado pelo RSpec.

Ainda podemos definir o timecop para operar no *safe mode*, em que é aceito apenas a sintaxe de bloco. Para isso, simplesmente adicionamos `Timecop.safe_mode = true` ao nosso `spec_helper.rb`.

4.4 Rails 4.1 e o ActiveSupport::Testing::TimeHelpers

A partir do Rails 4.1, foi criado o módulo `ActiveSupport::Testing::TimeHelpers`, que nos oferece métodos para viajarmos no tempo assim como com o timecop.

Como estamos utilizando o RSpec, primeiro temos que incluir o módulo. Para isso, utilizamos o `spec_helper.rb`.

```
RSpec.configure do |config|

  # ...
  config.include ActiveSupport::Testing::TimeHelpers
end
```

Com o módulo inserido simplesmente trocamos de `Timecop.freeze` para `travel_to` e de `Timecop.return` para `travel_back`, e mantemos o mesmo comportamento do timecop. No entanto, agora não há necessidade de uma gem extra.

```
before do
  hoje = Time.zone.local(2010, 3, 3, 12)
  travel_to(hoje)
end

after do
  travel_back
end
```

Assim como o `Timecop.freeze`, o `travel_to` também aceita um bloco, de forma que não é necessário usar o `travel_back`. No entanto, não se esqueça de sempre usar o `travel_back` se não estiver usando um bloco, como no nosso exemplo, para evitarmos o problema de testes quebradiços que vimos anteriormente.

Mas e o timecop ainda faz sentido? Sim! Em projetos que não são Rails ou que não utilizem o ActiveSupport. A dica é: se tiver em uma app Rails, ou se seu projeto tiver o ActiveSupport, utilize o `travel_to`; nos demais casos utilize o timecop.

4.5 Conclusão

Espero que tenha curtido nossa viagem no tempo! Agora temos em nosso arsenal mais uma ferramenta que nos ajuda a lidar com testes que dependam de data. Neste capítulo:

- Vimos que devemos ter cuidado ao lidarmos com testes que dependam de datas;
- Congelamos o tempo utilizando o timecop;
- Removemos repetição de nosso código utilizando o `let`;
- Vimos que tantos `let`s podem ser um sinal de que fizemos algo de errado;
- Utilizamos o timecop sem a sintaxe de bloco;
- Aprendemos que sempre devemos voltar ao tempo atual quando utilizamos o timecop sem o bloco;
- Conhecemos o `ActiveSupport::Testing::TimeHelpers`, que é uma alternativa ao timecop para quando estamos em um ambiente Rails ou que possua o ActiveSupport.

Agora que voltamos da nossa viagem no tempo, continue aí que no próximo capítulo conheceremos o SimpleCov, uma ferramenta que nos permite medir o que foi testado em nosso código, e veremos se devemos ou não ter 100% do nosso código coberto por testes.

Capítulo 5

Será que testei tudo?

5.1 Introdução

Durante o nosso dia a dia, estamos fazendo TDD enquanto escrevemos nossos códigos, no entanto, fica uma dúvida: será que testei tudo? Será que testei todas as possibilidades? Anteriormente vimos 1.6 que não devemos testar apenas o *happy path* mas como saber o que testamos e o que não testamos?

Conhecendo o SimpleCov

O SimpleCov é uma ferramenta de cobertura de código em Ruby que analisa quais linhas do código foram testadas e quais não, e nos retorna um HTML com o resultado gerado.

Instalamos o SimpleCov adicionando ao nosso Gemfile no grupo de testes.

```
gem 'simplecov', require: false
```

Para finalizar, incluímos o SimpleCov no `spec_helper.rb` e o iniciamos com `SimpleCov.start`. Vale lembrar que estas **linhas do SimpleCov devem ser a primeira coisa do nosso `spec_helper.rb`. Elas devem ser inseridas antes de qualquer código da aplicação ser incluído**.

```
require 'simplecov'
SimpleCov.start 'rails'
```

No nosso exemplo, passamos para o `SimpleCov.start` o `'rails'`, dado que estamos em uma app rails. Deste modo, ele utiliza o profile do Rails que agrupa os nossos testes de controllers, models etc.

Agora ao executarmos nossos testes é gerada uma saída falando qual a nossa cobertura de testes atualmente. Algo como:

```
Coverage report generated for RSpec to /meuprojeto/coverage.
5 / 5 LOC (100.0%) covered.
```

É gerado um arquivo HTML com detalhes em `coverage/index.html` e, como boa prática, ignore no git a pasta coverage.

Figura 5.1: HTML gerado pelo SimpleCov

Figura 5.2: Detalhe de um único arquivo no SimpleCov

Por padrão, o SimpleCov é rodado todas as vezes que um teste é executado, sempre exibindo este output durante o nosso TDD, o que é bastante chato, afinal não estamos preocupados com isso enquanto escrevemos nossos testes. Para isso, podemos definir que o SimpleCov será executado apenas se uma variável de ambiente estiver definida, vamos chamá-la de `coverage` e colocar o código do SimpleCov dentro de um `if`.

```
if ENV['coverage'] == 'on'
  require 'simplecov'
  SimpleCov.start 'rails' do
    minimum_coverage 100
  end
end
```

Deste modo, ao rodarmos nossos testes, não teremos a saída do SimpleCov apenas se definirmos isso explicitamente utilizando `$coverage=on rspec spec`.

5.2 O FALSO 100%

No Ruby, utilizamos bastante macros, aqueles como o `scope` e o `validates` do Active Record. No entanto, estes macros são avaliados enquanto a classe simplesmente é incluída no ambiente de testes. Vamos pegar o nosso escopo declarado no capítulo 4 e adicionar uma validação em `nome` e `id_nacional`.

```
class Pokemon < ActiveRecord::Base

  validates :nome, :id_nacional,  presence: true
  scope :escolhidos_ontem, -> do
    where(escolhido_em:
      1.day.ago.midnight..Time.zone.now.midnight)
  end
end
```

Rodamos nosso teste e, ao verificarmos, vemos que está com 100% de cobertura, isso devido ao uso dos macros. Por isso, sempre devemos ter atenção ao adicionarmos novo código e não testarmos.

Aprendendo com os erros

Vimos que os macros podem nos dar uma falsa confiança de que tudo foi testado, mas não apenas os macros, os nossos testes mesmos podem nos dar essa impressão.

No capítulo 1.6 falamos sobre a importância de não testarmos apenas o *happy path*, no entanto, mesmo testando diversos cenários pode ser que não consigamos prever algo. Imagine que está navegando no seu app e, em algum momento, se depara com um pokémon sem nome na tela, ou seja, algum caso de borda que não testamos. O que podemos fazer é utilizar uma ferramenta de debug, entender e replicar o problema, no navegador mesmo. Quando está claro qual o nosso problema, criamos um teste de regressão, um teste que recebe os mesmos parâmetros do problema, contudo, o seu resultado deve ser o esperado e não o erro.

Vamos pegar o nosso exemplo lá de quando falamos do *happy path*.

```
def nome_completo
  "#{nome} - #{id_nacional}"
end
```

Como vimos, pode ser que um pokémon esteja com o `nome` e o `id_nacional` vazios. Ao nos depararmos com um pokémon desse em nossa tela, teremos exibido simplesmente um `-` (traço com espaços no inicio e no final), o que não é nem de longe o que queremos. Uma abordagem rápida

porém errônea seria simplesmente adicionar o `if` ali para ele retornar `nil` e ser tratado corretamente na camada de apresentação.

```
def nome_completo
  "#{nome} - #{id_nacional}" if nome && id_nacional
end
```

No entanto, por que isso é errado? Porque não foi escrito nenhum teste, o famoso teste de regressão, afinal mudamos o comportamento de algo e não testamos. Assim outra pessoa do time pode vir durante um refactoring e simplesmente remover o `if` ao rodar os testes todos passaram, assim o problema foi inserido novamente no sistema.

O teste de regressão nada mais é do que um teste como outro qualquer. O mesmo teste escrito no capítulo 1.6 pode ser considerado um teste de regressão, se ele foi criado diante destas condições.

```
context 'quando não possui o nome e o id nacional' do

  subject do
    Pokemon.new
  end

  it 'é nil' do
    expect(subject.nome_completo).to be_nil
  end
end
```

Como regra é sempre bom adicionar uma linha, um `if` ou o que for escrever um teste, isso se estiver mudando o comportamento do objeto. Se for apenas um refactoring não é necessário, a menos que extraia código para uma nova classe.

Não teste associações, validações ou escopos do Active Record

Certa vez o criador do Rails fez um post sobre testes em que ele incluía 7 coisas a não se fazer enquanto se está realizando testes [3]. Uma delas era exatamente isto: não testar associações, validações ou escopos. Mas será que isso é uma coisa realmente boa a se fazer?

Os defensores desta maneira de realizar os testes defendem que este comportamento é testado pelo Rails, o que é correto. No entanto, o Rails não testa que eu chamei a declaração de uma associação ou validação, por exemplo. Assim, se meu modelo `Pokemon` definir as seguintes validações:

```
class Pokemon < ActiveRecord::Base

  validates :name, :id_nacional,  presence: true
end
```

E eu as remover, nenhum teste meu quebrará! Acabamos de ver o mal que isso pode trazer.

Alguns defendem que tais validações ou associações devem ser testadas em testes de controller ou de integração. Testes de integração são lentos, e por isso não são criados muitos casos de uso como de testes unitários. E no seu teste de controller, você realmente vai ficar testando cada um destes parâmetros? Ou vai testar apenas com um contexto válido ou um inválido? Este tipo de abordagem fica pior ainda ao falarmos de associações ou escopos.

Associações `has_many`, por exemplo, podem definir uma opção chamada `dependent`. Sendo assim, ao testar o deletar de um usuário no controller você vai testar também que foi deletado seus artigos? Em um simples blog, esta lógica é simples. Agora aumente a quantidade de relacionamentos que um modelo em um sistema de médio/grande porte possui. Se neste caso fizer os testes no controller, já é melhor do que assumir que é testado pelo Rails.

Escopos sofrem do mesmo problema de associação, definem um comportamento que pode ser complexo. Imagine o exemplo que utilizamos em 4. Será que no nosso controller teríamos escrito diversos casos de ponta ou apenas um simples teste para o escopo? Provavelmente apenas um simples caso, o *happy path*.

Uma das vantagens de escrevermos estes testes diretamente no model é que se alteramos algo nele, o teste do model quebrará e não o teste do controller, o que facilita na hora de ver o que foi quebrado.

Como sempre, nada em software é absoluto: temos os que não testam estes comportamentos [3] unitariamente, assim como os que testam. Estes

até criaram uma gem para ajudar a realizar alguns destes testes que veremos mais à frente. O importante aqui é testar, o que não pode é assumir que tal comportamento já é testado pelo Rails.

5.3 Meu objetivo é ter 100% de cobertura de testes?

Assim como temos escolas diferentes no modo como se realizar os testes, o mesmo ocorre quando o assunto é cobertura de testes. No post do DHH [3], uma de suas práticas é que não devemos focar em obter 100% de cobertura de testes. Na outra ponta temos Uncle Bob [5] que acredita que cada linha de código que você escreve deve ser testada.

O importante é o time ver valor ao escrever os testes. Não adianta um gerente ou alguém assim impor esta prática ao time. Como vimos, podemos ter o falso 100%, e caso o time não esteja motivado a realizar os testes, é bem provável que escrevam testes bem ruins para simplesmente agradar a ferramenta de cobertura de testes. Por isso, é importante o time definir qual será esta cobertura, e adicionar a ferramenta.

Mesmo um projeto que tenha 100% de cobertura de testes pode ter bugs, pois, como vimos, pode ser um caso de falso 100%. No entanto, se o time está escrevendo testes porque vê valor, e não para agradar a ferramenta, é bem provável que a quantidade de bugs seja reduzida. Contudo, lembre-se de que no projeto que não possui estes 100% de cobertura a chance de ter bugs são maiores.

Um ganho ao se ter uma alta taxa de cobertura de testes é na hora em que iremos realizar refatorações, afinal temos a confiança de que podemos remover um método, alterar seu comportamento, extrair classes etc. Ao realizarmos um movimento errado durante a refatoração, algum teste quebrará e nos alertará para o problema, de modo que temos total confiança ao fazer mudanças para melhorar o nosso código. Em projetos com uma taxa de cobertura menor, provavelmente ninguém irá alterar aquele método do model que não possui testes unitários. Se for alterá-lo, é melhor escrever testes unitários para ele para só depois fazer a refatoração. Assim garantimos que não quebramos seu comportamento.

Outro benefício de uma alta taxa de cobertura de testes é que ela nos ajuda a encontrar código que não é mais utilizado e, no entanto, ainda está ali. Normalmente, quando estamos fazendo refatorações movendo métodos de um lado para outro, definindo alguns como privados e removendo outros, é bem provável que tenha algo ali que não é mais utilizado. Com 100% de cobertura de testes, conseguimos ter isso facilmente exposto, dado que podemos ter no nosso processo de integração esta verificação de 100% de cobertura e só mandar código para o repositório central se o código atingir tal nível.

Assim, quando fôssemos integrar nosso código, a verificação de cobertura de testes falharia e, ao verificarmos o porquê, encontraríamos um método privado que não é mais utilizado, e simplesmente o removeríamos.

Um dos pontos que pode pesar na hora de se definir uma meta de cobertura de código é o custo, afinal, agora é necessário cada desenvolvedor escrever mais para entregar a mesma funcionalidade. No mesmo post do DHH, ele cita um trecho do Kent Beck. Em uma tradução livre:

"Sou pago para o código que funciona, não para testes, por isso a minha filosofia é testar o mínimo possível para chegar a um determinado nível de confiança (eu suspeito que este nível de confiança é alto em comparação com os padrões da indústria, mas poderia ser apenas arrogância). Se eu não costumo fazer um tipo de erro (como definir as variáveis erradas em um construtor), eu não o testo."
– Kent Beck

Quem sou eu para discordar do Kent Beck? =)

Mas um detalhe aqui é quando ele fala "Se eu não costumo fazer um tipo de erro". Normalmente, não estamos trabalhando sozinhos, mas em conjunto, em times multidisciplinares e com código coletivo. Sendo assim, um novo membro do time com menos experiência pode cometer um erro que para você seja trivial, e assim inserir um bug no sistema que pode ser pego somente no ambiente de produção.

Por isso, acredito que os testes não são pagos nas primeiras semanas, mas sim com o passar do tempo. Afinal, com o passar do tempo o código tende a crescer e ficar mais complexo. Em um ambiente onde não se possui uma

grande cobertura de testes, começará a ficar confuso saber se alguma alteração terá um impacto negativo no sistema ou não, se um novo bug será inserido ou não. Com uma boa cobertura de testes, temos maior confiança ao lidar com o código, o que nos ajuda na mantenabilidade do projeto. Os testes funcionam como um remédio: podemos ir tomando-o em pequenas doses e constantemente, ou esperar estarmos em um estado bem ruim de saúde para só assim procurar um médico. Sabemos muito bem que o segundo caso tem uma dor bem maior.

5.4 Mas e você senhor autor, quanto faz de cobertura de testes?

Como já deve imaginar pelo o que acabou de ler, sim eu sou um dos defensores de cobertura de código. E é definido no processo de integração que, se não tiver 100% de cobertura, o código não é integrado. Agora vamos aos detalhes.

Primeiro de tudo, não pense em cobertura de testes enquanto faz o seu desenvolvimento com TDD. De preferência, defina que o SimpleCov seja executado por demanda, de modo que, enquanto roda seus testes durante o desenvolvimento, o output dele nem é exibido, como vimos anteriormente. Adicione ao seu processo de integração esta verificação de que se o código não estiver com 100% de cobertura de testes ele não será enviado. Pode ser que no começo tenha alguma dificuldade, mas com o tempo verá que terá menos de 100% de cobertura apenas quando está fazendo um *spike*, ou devido a código morto que pode ser removido.

O importante é criarmos o hábito de fazer isso no nosso dia a dia, assim como fazemos o teste: adicionar mais coisa no nosso jogo, que agora deve ser conseguir 100% de cobertura. Desde que vim para o Ruby sempre escrevo testes, inclusive foi um dos motivos para eu vir para trabalhar com Ruby. Com todo o ecossistema da comunidade Ruby, temos diversas ferramentas que nos ajudam a atingir esta meta, como as que vimos e as que ainda vamos ver.

Em projetos novos, defina no seu fluxo de integração a verificação de cobertura de 100% como falado anteriormente, assim ajudará a criar o hábito. Em projetos que já possuem uma história, adicione ao processo de integração a verificação, no entanto, com o valor da cobertura de testes do projeto, e vá

adicionando testes nas áreas que ainda não possuem. Em um projeto que está com 80% de cobertura, adicione que todo o código deve ter este mínimo para ser integrado. Com o tempo, esta taxa vai aumentar. Aumente novamente o mínimo de cobertura do projeto para ele ser integrado, e em um momento é possível chegar aos 100%.

Realmente recomendo que tenhamos 100% de cobertura de testes, dado que temos diversos benefícios que a meu ver superam os problemas relatados.

Jumpup

Falamos muito do processo de integração, contudo, nenhuma ferramenta foi citada. Isso porque você pode está fazendo integração assíncrona utilizando o Travis CI, por exemplo, ou ainda fazendo integração síncrona.

Caso esteja realizando integração síncrona, recomendo o uso da gem Jumpup, que faz esta verificação de cobertura de testes e só envia código para o repositório central se este atingir 100%.

https://github.com/Helabs/jumpup

5.5 Conclusão

Vimos diversas opiniões que a comunidade possui quando o assunto é cobertura de testes, vamos agora ao resumo.

- Conhecemos o SimpleCov;
- Utilizamos o SimpleCov por demanda;
- Aprendemos que podemos ter um falso 100% de cobertura de código;
- Passamos por testes de regressão;
- Vimos o famoso "não teste associações, validações ou escopos" do Active Record;
- Falamos sobre o 100% de cobertura de testes;

- Finalizamos vendo como eu pessoalmente lido com o 100% de cobertura de testes.

Testes de validações de model de um app rails tendem a acabar ficando bem repetitivos entre diversos models. No próximo capítulo veremos como podemos remover esta repetição começando por um *shared example* e, em seguida, um matcher, conhecendo alguns matchers de terceiros que podem nos ajudar nestas tarefas.

Capítulo 6

Copiar e colar não é uma opção!

6.1 Introdução

Vamos adicionar validação ao nosso model `Pokemon`, pois até o momento ele pode ser salvo com todos os valores `nil` e não é o que queremos. Para isso, vamos começar pelo teste, criando um caso que deve dar erro ao estar com o valor de nome vazio. Instanciamos um Pokémon e chamamos o método `valid?`, que roda todas as validações de um model. Depois das validações executadas, verificamos no array `errors` se a nossa chave `nome` possui algum erro.

```
describe 'validações' do

  describe '#nome' do

    it 'possui erro quando está vazio' do
```

```
      pokemon = Pokemon.new
      pokemon.valid?
      expect(pokemon.errors[:nome]).
        to include('não pode ficar em branco')
    end
  end
end
```

Rodamos o nosso teste e ele quebra. Vamos agora ao model implementar a nossa validação.

```
class Pokemon < ActiveRecord::Base

  validates :nome, presence: true
end
```

Executamos novamente o nosso teste, no entanto agora ele passa. Vamos testar agora que ao passarmos um nome para o nosso pokémon, ele não possua mais nenhum erro. Novamente, instanciamos um pokémon, agora definindo o seu nome, e verificamos o array `errors` após executarmos o `valid?`. Desta vez esperamos que ele possua um array vazio.

```
it 'não possui erro quando está preenchido' do
  pokemon = Pokemon.new(nome: 'Charizard')
  pokemon.valid?
  expect(pokemon.errors[:nome]).to be_empty
end
```

Assim finalizamos a validação de presença do campo nome no nosso model `Pokemon`. Agora temos que validar o campo `id_nacional`, sem falar de outros campos em outras entidades do nosso sistema que possuem a mesma validação de presença. Sabemos que copiar e colar o teste ali seria um problema, pois teríamos diversos testes duplicados por todo o sistema.

6.2 O shared example

O RSpec possui os *shared examples* que nos ajuda exatamente no que precisamos: criar testes reutilizáveis.

Vamos primeiro mover os nossos testes do model `Pokemon` referentes à validação do nome para um arquivo separado. Por convenção, os shared examples são armazenados em `spec/support/` e possuem o prefixo `shared_examples_for_`. Criaremos o nosso `spec/support/shared_examples_for_validacao.rb`. Para criarmos um shared example, utilizaremos o método `shared_examples`, que recebe como primeiro parâmetro o nome do nosso shared example e um bloco com o conteúdo do exemplo compartilhado.

```
shared_examples 'valida presenca de string' do

  describe '#nome' do

    it 'possui erro quando está vazio' do
     pokemon = Pokemon.new
     pokemon.valid?
     expect(pokemon.errors[:nome]).
        to include('não pode ficar em branco')
    end

    # ...
  end
end
```

Criamos o nosso shared exemple como `valida presenca de string` para focarmos apenas na validação de presença de campos do tipo string no momento. Agora que temos nosso shared example criado, devemos utilizá-lo no nosso model. Para isso utilizamos o `include_examples` passando o nome do shared example.

```
describe 'validações' do

  include_examples 'valida presenca de string'
end
```

Rodamos nossos testes e vemos que os exemplos continuam passando.

Sempre que tiver shared examples e quiser rodar estes testes, execute todo o arquivo de testes e não apenas um único teste. Isso porque os shared exam-

ples só funcionam ao executarmos o arquivo todo. No dia a dia, essa é uma boa dica para se manter em mente.

Mas ainda assim, olhando novamente para o nosso shared example, vemos que ele ainda não resolve nosso problema de duplicação, dado que ele apenas funciona com o model `Pokemon` e o atributo nome. Vamos agora ver como fazer isso de uma maneira dinâmica.

Shared examples dinâmicos

O shared example aceita parâmetros, sendo assim, temos que passar como parâmetro a classe e um símbolo com o nome do campo do model cuja presença queremos testar. Alteramos o nosso shared example para agora utilizar o atributo que foi passado por parâmetro. Além disso, instanciamos uma classe de acordo com a que foi passada por parâmetro e assim realizamos nossa validação de presença.

```
shared_examples 'valida presenca de string' do |klass, attr|

  describe "#{attr}" do

    it 'possui erro quando está vazio' do
      instancia = klass.new
      instancia.valid?
      expect(instancia.errors[attr]).
          to include('não pode ficar em branco')
    end
  end
end
```

Agora que fizemos o nosso primeiro teste ser dinâmico, vamos fazer a mesma coisa no segundo. Criamos um hash dinamicamente, definindo o valor do atributo passado como Charizard. Depois de executarmos as validações da nossa classe, verificamos se o atributo passado não possui nenhum erro.

```
it 'não possui erro quando está preenchido' do
  params = {}
  params[attr] = 'Charizard'
```

```
  instancia = klass.new(params)
  instancia.valid?
  expect(instancia.errors[attr]).to be_empty
end
```

Agora que definimos o nosso shared example de uma maneira dinâmica temos que alterar o nosso teste para passar os parâmetros corretos. Simplesmente passamos a nossa classe `Pokemon` e um símbolo `:nome`, que é o valor que queremos testar.

```
describe 'validações' do

  include_examples 'valida presenca de string', Pokemon, :nome
end
```

Nosso shared example agora pode ser usado em outros campos do model `Pokemon`, assim como em outras entidades que necessitem de validação de presença em um campo string.

Poderíamos ir além e alterar o nosso exemplo para ele funcionar baseado no campo definido e o preenchimento do valor seja feito automático, independente de ser uma string. Deixo isso como um exercício para você, leitor.

Como definimos a descrição do nosso teste dinamicamente, ainda continuamos com um output excelente quando rodamos os testes com o formato de documentação.

```
Pokemon
  validações
    nome
      possui erro quando está vazio
      não possui erro quando está preenchido
```

6.3 Criando um Matcher

Ao instalarmos o RSpec, já temos uma coleção de matchers padrões, que são definidos na gem `rspec-expectations`, uma depêndencia do RSpec. O `rspec-expectations` é quem define os matchers que estamos acostumados a utilizar como `eq`, `include` e assim por diante. Além dos matchers

padrões, o RSpec nos dá a possibilidade de criarmos o nosso próprio matcher, então vamos mover o nosso shared example para um matcher.

Antes de iniciarmos o código, vamos definir a interface do nosso matcher. No matcher `eq`, pegamos o resultado e comparamos com o esperado.

```
it 'soma dois inteiros' do
  expect(soma(2,2)).to eq(4)
end
```

No nosso caso, eu não quero ter que executar a validação. No lugar disso, passaremos apenas a classe que será testada. E como no matcher eu quero passar apenas o atributo, nossa interface tem que ser usada assim:

```
it { expect(Pokemon).to valida_presenca_de_string(:nome) }
```

Agora que já temos uma interface definida, vamos criar o nosso matcher. Para isso, começamos com o `RSpec::Matchers.define` e passamos para ele como primeiro parâmetro o nome do nosso matcher. Em seguida, passamos um bloco e como parâmetro dele o valor que será avaliado no matcher.

Temos que implementar o `match` que também recebe um bloco cujo parâmetro é o que foi passado para o `expect`, no caso o `sujeito`. É aqui que definimos se a condição foi válida ou não retornando `true` ou `false`.

```
RSpec::Matchers.define :valida_presenca_de_string do |attr|

  match do |sujeito|
  end
end
```

Com o básico pronto, agora temos que mover a nossa lógica do shared example para cá. Para isso criamos um método `verifica_vazio?` que possui o conteúdo bem parecido do nosso shared example no teste "`possui erro quando está vazio`", exceto que agora ele é um método e que não fazemos asserção nenhuma dentro dele. Ele deve apenas retornar `true` ou `false`.

```
match do |sujeito|
  verifica_vazio?(sujeito, attr)
```

```
end

def verifica_vazio?(sujeito, attr)
  instancia = sujeito.new
  instancia.valid?
  instancia.errors[attr].include?('não pode ficar em branco')
end
```

Implementamos a primeira parte do nosso teste, mas não podemos esquecer do outro. Movemo-lo para cá e, igual ao anterior, retornamos apenas o resultado, sem nenhuma asserção. Finalizados ambos os métodos, chamamos um seguido do outro dentro do match, utilizando o `&&`.

```
match do |sujeito|
  verifica_vazio?(sujeito, attr) &&
  verifica_preenchido?(sujeito, attr)
end

def verifica_preenchido?(sujeito, attr)
  params = {}
  params[attr] = 'Charizard'
  instancia = sujeito.new(params)
  instancia.valid?
  instancia.errors[attr].empty?
end
```

Assim, finalizamos a estrutura do nosso matcher e podemos utilizá-lo conforme definimos a assinatura.

```
it { expect(Pokemon).to valida_presenca_de_string(:nome) }
```

Por uma melhor descrição

No momento, se rodarmos nosso teste e ele falhar teremos uma mensagem gerada pelo próprio RSpec.

```
expected Pokemon(id: integer, id_nacional: integer, nome: string,
ataque: integer, defesa: integer, created_at: datetime,
updated_at: datetime) to valida presenca de string :nome
```

Não está muito claro o problema, já que ele mostrou todos os campos da classe, além de exibir um símbolo no nome do atributo. Podemos melhorar esta saída utilizando o método `failure_message`, que recebe um bloco em que passamos o nosso sujeito e mostramos uma mensagem customizada.

```
failure_message do |sujeito|
  "esperava-se que #{sujeito} tivesse validação em #{attr}"
end
```

Agora que definimos o `failure_message`, teremos a seguinte saída em caso de falha.

```
esperava-se que Pokemon tivesse validação de presença em nome
```

Uma coisa legal para se lembrar ao criar um matcher é que ele funciona também em caso de negação, ao utilizarmos o `to_not`, o que nos dá bastante flexibilidade. Matchers também geram bastante expressividade em nosso código.

Assim como utilizamos o nosso matcher com o `#to`, haverá casos em que queremos utilizá-lo com o `to_not`, porém, ficaria estranha a saída se tivéssemos a mesma mensagem que definimos no `failure_message`, afinal estamos fazendo o oposto agora.

Para definirmos a mensagem para quando estivermos utilizando o `to_not`, temos o método `failure_message_when_negated` que funciona exatamente como `failure_message`, onde apenas definimos uma mensagem que faça sentido em caso de negação.

```
failure_message_when_negated do |sujeito|
  "esperava-se que #{sujeito} não tivesse validação em #{attr}"
end
```

Uma das vantagens de termos matchers bem específicos é podermos usar o matcher sem a necessidade de escrevermos uma mensagem para o `it`, dado que o próprio matcher já diz o comportamento esperado.

Isso não quer dizer que não devemos utilizar do `it` com uma mensagem, na realidade devemos sempre definir uma boa mensagem que deixe claro o comportamento testado. Utilizamos o matcher de uma linha apenas quando

este define claramente o comportamento testado e possua uma boa saída no formato de documentação.

O RSpec define uma descrição automaticamente quando criamos o nosso matcher, mas podemos melhorar esta saída. Para isso utilizamos o método `description` e passamos para ele apenas a nossa mensagem que queremos exibir.

```
description do
  "valida presença do #{attr}"
end
```

Assim, finalizamos o nosso próprio matcher.

Agora que criamos nosso matcher, não quer dizer que o nosso shared example não faça sentido, depende do desenvolvedor decidir qual usar. Um bom caminho é se perguntar se aquela extração que está sendo feita será utilizada em mais de uma entidade, por exemplo, se é algo realmente genérico. Mas como regra, podemos seguir utilizando o shared example e extrair para um matcher apenas quando sentirmos necessidade.

Uma vez que aprendemos como criar nossos próprios matchers, é comum querermos encapsular diversos deles em uma gem para facilitar nosso trabalho e ainda contribuir com a comunidade... mas calma ae!

Já temos diversas coleções de matchers encapsuladas em gems e mantidas pela comunidade. Antes de criar um novo, é melhor ver se o que você está pensando fazer já não existe. Mesmo se não existir um matcher, é mais fácil criá-lo em um destes projetos do que criar um novo projeto do zero.

6.4 O SHOULDA-MATCHERS

O shoulda-matchers foi extraido do shoulda, que é uma meta gem que é composta do shoulda-context e do shoulda-matchers. O shoulda-context é utilizado para escrever testes mais legíveis no `Test::Unit`. Com esta extração, podemos utilizar apenas os matchers, que é o que realmente nos interessa. Os matcher criados pelo shoulda-matchers são para testarmos funcionalidades comuns do Rails.

Para instalarmos o shoulda-matchers, simplesmente o adicionamos ao Gemfile no grupo de testes e rodamos `$ bundle`.

```
group :test do
  gem 'shoulda-matchers'
end
```

Vamos agora validar o nosso model `Pokemon`, que deve possuir validação de presença nos campos `nome` e `id_nacional`. Para isso, utilizaremos o matcher `validate_presence_of`, que recebe como parâmetro o nome da coluna do banco cuja presença estamos validando.

```
describe 'validações' do

  it { should validate_presence_of(:nome) }
end
```

Assim como o nosso matcher que criamos, o shoulda-matchers possui uma boa descrição e podemos o utilizar sem a necessidade de definir uma descrição ao teste. Diferente do matcher que criamos anteriormente, o `validate_presence_of` funciona com todos os tipos de dados, então vamos validar que a nossa coluna `id_nacional` deve ser preenchida também.

```
it { should validate_presence_of(:nome) }
it { should validate_presence_of(:id_nacional) }
```

E para nossos testes passarem, simplesmente adicionamos a validação de presença no model.

```
class Pokemon < ActiveRecord::Base

  validates :nome, :id_nacional, presence: true
end
```

Este é apenas um começo do que o shoulda-matchers é capaz. Vamos continuar a validação do nosso model para vermos mais dos seus poderes.

Matchers com superpoderes

Nosso campo `id_nacional` não deve aceitar qualquer valor, ele deve aceitar apenas valores inteiros e maiores que o. Vamos primeiro testar que ele aceita apenas numeros, para isso utilizamos o `validate_numericality_of`

```
it { should validate_numericality_of(:id_nacional) }
```

E para fazer o teste passar apenas adicionamos validação de valores numéricos utilizando o parâmetro `numericality`.

```
class Pokemon < ActiveRecord::Base

  validates :nome, :id_nacional, presence: true
  validates :id_nacional, numericality: true
end
```

Agora temos que garantir que aceitamos apenas valores inteiros e não qualquer valor. O shoulda-matchers nos dá a possibilidade de utilizar métodos encadeados e ele já possui um método que podemos passar para o matcher para garantir que aceitamos apenas valores inteiros o método `only_integer`.

```
it { should validate_numericality_of(:id_nacional).only_integer}
```

No nosso model, para o teste passar, simplesmente adicionamos um hash passando a opção de apenas inteiro.

```
validates :id_nacional, numericality: { only_integer: true }
```

Para finalizar a validação do nosso `id_nacional`, temos que garantir que todo o valor seja acima de zero. Como você já deve imaginar, podemos encadear mais um método ali que resolve o nosso problema, o `is_greater_than`, que recebe como parâmetro o valor que o model deve aceitar acima deste.

```
it { should validate_numericality_of(:id_nacional).only_integer
        .is_greater_than(0) }
```

E para o nosso teste voltar a passar, simplesmente adicionamos a chave `greater_than` no nosso model.

```
class Pokemon < ActiveRecord::Base

  validates :nome, :id_nacional, presence: true
  validates :id_nacional, numericality: {
    only_integer: true, greater_than: 0 }
end
```

Como se pode ver, temos uma sintaxe bem flexível para realizar os nossos testes. É bom sempre fazermos o TDD mesmo quando utilizamos os matchers como fizemos anteriormente, pois com o uso de matchers podemos querer "agilizar" as coisas, mas vimos que mesmo com o uso de matchers é possível fazermos TDD.

Para finalizar, vamos organizar os nossos testes separando por contexto cada uma das validações dos campos.

```
describe 'validações' do

  it { should validate_presence_of(:nome) }

  describe 'id_nacional' do

    it { should validate_presence_of(:id_nacional) }
    it { should validate_numericality_of(:id_nacional).
        only_integer.is_greater_than(0) }
  end
end
```

Definindo o sujeito nos matchers

Nosso model `Pokemon` deve exigir a presença do campo ataque, no entanto apenas quando o valor da coluna aprovado for `true`, ou seja, temos uma validação condicional. Isto porque não queremos que um pokémon seja aprovado por engano quando não possui o valor do seu ataque.

Utilizar o shoulda-matchers com o sujeito implícito como fizemos anteriormente não resolverá nosso problema. Teremos como sujeito um pokémon

com todos os campos `nil`, mas temos que definir o nosso sujeito e definir o valor da coluna aprovado.

Para isso, iniciaremos criando um contexto de quando o pokémon está aprovado e, nesse contexto, ele deve validar a presença do nome.

```
describe 'ataque' do

  context 'quando está aprovado' do

    subject do
      Pokemon.new(aprovado: true )
    end

    it { should validate_presence_of(:ataque) }
  end
end
```

Para fazer o nosso teste passar, simplesmente adicionamos a validação de presença no nosso model.

```
validates :ataque, presence: true
```

Testamos o caso de quando o pokémon está aprovado, agora temos que testar quando ele não está aprovado. Para isso, criamos um contexto e testamos que não temos validação de presença caso o pokémon não tenha definido o aprovado para `true`.

```
describe 'ataque' do

  # ...

  context 'quando não está aprovado' do

    subject do
      Pokemon.new
    end

    it { should_not validate_presence_of(:ataque) }
  end
end
```

Ao rodarmos nosso teste, ele quebra, como o esperado. Para ele passar, simplesmente passamos a opção `:if` com o valor `aprovado?`, dado que o Active Record implementa para nós este método que verifica se o valor está preenchido ou não.

```
validates :ataque, presence: true, if: :aprovado?
```

Assim, finalizamos a nossa introdução ao shoulda-matchers. E como se pode ver agora ficou bem mais fácil de testar a validação de presença do campo nome, e que a idade é realmente um inteiro positivo em um model `Usuario`, que é caso comum em muitas aplicações.

Fica a dica para consultar a documentação do shoulda-mathcers para ver os demais matchers disponíveis. Mas apenas para deixar um gostinho de quero mais, lá temos matchers para testar associações do Active Record, diversos matchers de validação, além dos de controller para testar renderização de layouts, templates e matchers de rotas. Sempre que estiver testando algo comum, não se esqueça de dar uma olhada no shoulda-matchers para verificar se já não existe um matcher para o que está testando.

https://github.com/thoughtbot/shoulda-matchers

6.5 Matchers de terceiros

O shoulda-matchers não é a única coleção de matchers, temos coleções de matchers para diversos testes comuns que nossa aplicação possa ter. Vamos apenas conhecer alguns deles, dado que todos funcionam bem parecido como o shoulda-matchers. Fica como exercício utilizá-los nos projetos em que fizer sentido.

- **email-spec:** como o nome já diz, ajuda-nos a testar os e-mails do ActionMailer https://github.com/bmabey/email-spec.
- **rspec-sidekiq:** para quando estamos utilizando o Sidekiq (ferramenta de *background job*) https://github.com/philostler/rspec-sidekiq.
- **mongoid-rspec:** assim como o shoulda-matchers nos dá diversos matchers do Active Record, o mongoid-rspec nos oferece matchers para

quando estamos utilizando o Mongoid https://github.com/evansagge/mongoid-rspec.

Estes são apenas alguns. Quando ver que está testando uma coisa que é bem comum e repetitiva, vale a pena procurar para ver se não é algo que já possui uma gem que facilitará o nosso trabalho.

Além das gems que são coleções de matchers, temos gems que já possuem os seus matchers, como é o caso do Paperclip. Por isso, é sempre bom ler a documentação da gem que estamos usando para não perdemos estas dicas.

Nenhum matcher vai resolver todos os nossos problemas. Ainda teremos que fazer o nosso TDD. O papel do matcher é apenas nos facilitar em testes repetitivos. Sendo assim, se ainda não se sentir confortável ao realizar TDD, continue fazendo o TDD para só depois utilizar estes matchers de terceiros, dado que bastante coisa é testada ao utilizarmos os matchers.

6.6 Conclusão

Vimos diversas maneiras de evitar repetição nos nossos testes:

- Extraímos nossos testes repetidos para um shared example;
- Utilizamos shared examples dinâmicos;
- Criamos o nosso próprio matcher;
- Melhoramos as saídas do nosso matcher;
- Conhecemos o shoulda-matchers;
- Fomos apresentados a mais alguns matchers de terceiros.

Como podemos testar valores não determinísticos? Como fazer um teste de uma classe que utiliza outra que ainda nem existe? Fica por aí que no próximo capítulo veremos bastante coisa legal no uso de mocks e stubs que nos ajudam resolver estes problemas.

Capítulo 7

O tal dos mocks e stubs

Nosso model `Pokemon` agora precisa retornar um valor para quando um ataque é crítico. Como estamos apenas no começo do projeto, o valor do nosso ataque crítico deve ser um valor aleatório entre 60 e 80. Mas como podemos testar algo que não possui um valor não determinístico? Afinal, em um momento o valor retornado pode ser 66 e no outro 75.

7.1 Conhecendo o stub

O RSpec já possui no seu core uma biblioteca de mocks, o rspec-mocks, que é exatamente o que precisamos quando queremos forjar a chamada de algum método.

Antes de realizarmos nosso primeiro uso do stub, vamos primeiro definir o RSpec para utilizar apenas a sintaxe de `expectativa` quando estivermos

fazendo nossos mocks. Para isso, utilizamos a configuração `mock_with` no nosso `spec_helper.rb`.

```
RSpec.configure do |config|
  # ...

  config.mock_with :rspec do |mocks|
    mocks.syntax = :expect
  end
end
```

Com o RSpec devidamente configurado, vamos ao nosso primeiro teste utilizando stub, que nos permite forjar a chamada de algum método.

Utilizando o stub

Vamos começar o teste do nosso método `Pokemon#ataque_critico`. Para isso, criamos um contexto chamado `#ataque_critico` e dentro dele criamos o teste do valor aleatório, testando apenas que o valor deve ser igual a 75.

```
describe '#ataque_critico' do

  it 'é um valor aleatório' do
    pokemon = Pokemon.new
    expect(pokemon.ataque_critico).to eq(75)
  end
end
```

Vamos agora implementar o nosso método `ataque_critico`. No Ruby podemos utilizar a classe `Random` e seu método de classe `rand`, que recebe um range e nos retorna um valor aleatório dentro daquele range. Sendo assim, podemos passar um range entre 60 e 80, que nosso problema está resolvido.

```
class Pokemon < ActiveRecord::Base

  def ataque_critico
    Random.rand(60..80)
  end
end
```

Vamos rodar os testes. Como é de se esperar, os testes quebram. Em algum momento ele vai passar, mas também quebrará em muitos, devido à dinâmica do `Random`. Não podemos ter este tipo de comportamento em nossa suíte de testes, portanto, vamos agora utilizar o stub para resolver o nosso problema.

O que queremos agora é forjar a chamada do nosso objeto colaborador, o `Random`, e seu método `.rand`. Para isso, utilizamos o `allow`, que recebe a classe ou objeto em que iremos fazer o stub. Em seguida, passamos para o método `receive` o método que queremos forjar e, para finalizar, o valor retornado por este método com o uso do `and_return`.

```
it 'é um valor aleatório' do
  allow(Random).to receive(:rand).and_return(75)
  pokemon = Pokemon.new
  expect(pokemon.ataque_critico).to eq(75)
end
```

Agora sempre que rodarmos nosso teste o valor de `Random.rand` será de 75 e nosso teste sempre passará.

Asserção dos parâmetros

Nosso teste passa, no entanto, se formos lá no model `Pokemon` e alterarmos `Random.rand` para não receber nenhum parâmetro ou outro parâmetro qualquer, o nosso teste continuará passando, o que não é o esperado.

Ao utilizarmos stub, temos a opção de chamar o método encadeado `with`, que nos permite fazer uma asserção nos parâmetros que são esperados, os que o método deve receber. Para isso, simplesmente passamos os parâmetros corretos depois do uso do `receive` com o `with`.

```
it 'é um valor aleatório' do
  allow(random).to receive(:rand).with(60..80).and_return(75)
  pokemon = pokemon.new
  expect(pokemon.ataque_critico).to eq(75)
end
```

Agora se formos ao nosso model e alterarmos a chamada ao `Random.rand`, o teste falhará, dado que a chamada mudou.

WTF! Eu não testei nada

Calma aí! O conceito de stub muitas vezes nos dá a sensação de que não testamos nada, afinal estamos forjando a chamada de um método. Mas sabe quem também são forjadores que vimos aqui? O WebMock, que forja chamadas HTTP, o Timecop, que forja o tempo, e o `build_stubbed` do FactoryGirl, que forja um objeto Active Record. Observe que todos eles forjam colaboradores de nosso sistema, nunca devemos forjar o objeto testado. Note que na realidade não forjamos o nosso método `Pokemon#ataque_critico`, que é o nosso método testado, e sim o seu colaborador que é o `Random.rand`.

Vamos agora às dicas de quando utilizar stub.

- Quando o resultado de um dos seus colaboradores não é determinístico;
- Apenas em colaboradores, nunca no objeto (o sujeito), do seu teste;
- Quando o colaborador faz uma operação lenta, como acessar uma API.

7.2 Os dublês

Temos em nossas mãos agora a tarefa de criar um *Card* para os pokémons, para os usuários e qualquer outra entidade do nosso sistema, desde que ela implemente o método `to_presenter`. No entanto, ainda não foi implementado nenhum destes métodos `to_presenter` em nenhuma dessas entidades, então como podemos testar se o nosso colaborador ainda não existe? Para estes casos, existem os dublês do RSpec.

O contrato entre o `CardPresenter` e o colaborador é que o colaborador deve implementar o método `to_presenter` que retorne um hash com a chave sendo o atributo que quer exibir no card e o valor do hash como o valor para aquele atributo.

Vamos começar criando o nosso teste para o método `CardPresenter#show`. Iniciamos instanciando um `CardPresenter` passando um objeto e fazemos a nossa expectativa de que ao chamar o método `#show` devemos ter um parágrafo para cada atributo.

```
describe '#show' do

  it 'retorna um paragrafo por chave' do
    card_presenter = CardPresenter.new(objeto)
    expect(card_presenter.show).
      to eq(%{<p>nome: Mauro</p><p>idade: 24</p>})
  end
end
```

Agora vamos criar o nosso dublê, o `objeto`. Para isso, utilizamos o método `double` e passamos para ele uma descrição. É importante passar a descrição pois isso nos ajuda na saída do nosso teste.

```
it 'retorna um paragrafo por chave' do
  objeto = double('Um objeto')
  card_presenter = CardPresenter.new(objeto)
  expect(card_presenter.show).
    to eq(%{<p>nome: Mauro</p><p>idade: 24</p>})
end
```

Neste momento já estamos passando o nosso dublê para o `CardPresenter`, porém o nosso dublê não implementa o método `to_presenter`. Lembra dos stubs? Então vamos agora criar um stub para o nosso dublê, utilizando o `allow`, mas com a diferença de que agora estamos passando um objeto, o nosso dublê, e não uma classe como fizemos anteriormente.

```
it 'retorna um paragrafo por chave' do
  objeto = double('Um objeto')
  to_presenter = { nome: 'Mauro', idade: 24 }
  allow(objeto).to receive(:to_presenter).
    and_return(to_presenter)
  card_presenter = CardPresenter.new(objeto)
  expect(card_presenter.show).
    to eq(%{<p>nome: Mauro</p><p>idade: 24</p>})
end
```

Agora temos o nosso cenário montado. Temos o colaborador, que é o `objeto`, e o nosso sujeito, o `CardPresenter`, que consegue fazer sua asserção. Vamos agora escrever a nossa classe `CardPresenter`. Começamos

criando um initializer que cria um `@objeto`. Definimos também o acesso a este objeto para apenas leitura dentro do `CardPresenter` utilizando o `attr_reader`.

```
class CardPresenter

  def initialize(objeto)
    @objeto = objeto
  end

  private

    attr_reader :objeto
end
```

E para finalizar, criamos o método `CardPresenter#show` que simplesmente itera em cada um dos valores do hash do `to_presenter` e monta um parágrafo.

```
def show
  retorno = ''
  objeto.to_presenter.each do |atributo, valor|
    retorno += "<p>#{atributo}: #{valor}</p>"
  end
  retorno
end
```

Ao rodarmos o nosso teste neste momento, ele passará. Isso porque o nosso dublê implementa o método `to_presenter` retornando um hash.

Como vimos, um dublê é um objeto que simula outros objetos e que pode ter seu comportamento alterado de forma controlada com o uso de stub.

Para finalizar o nosso teste, vamos apenas fazer um refatoração para deixar mais claro o papel de cada um. Movemos o nosso colaborador para um `let`, definimos o nosso sujeito utilizando o `subject`, utilizamos o bloco `before` para fazermos o nosso stub e, para finalizar, fazemos o nosso teste.

```
describe '#show' do
```

```
  let(:objeto) do
    double('Um objeto')
  end

  subject(:card_presenter) do
    CardPresenter.new(objeto)
  end

  before do
    to_presenter = { nome: 'Mauro', idade: 24 }
    allow(objeto).to receive(:to_presenter).
      and_return(to_presenter)
  end

  it 'retorna um paragrafo por chave' do
    expect(card_presenter.show).
      to eq(%{<p>nome: Mauro</p><p>idade: 24</p>})
  end
end
```

Como pode ver, fica mais claro o que está acontecendo, isso devido à expressividade que o RSpec nos fornece com sua DSL.

Os super dublês

Nosso teste funciona, no entanto, nada garante que ele funciona com o `Pokemon`, dado que utilizamos um dublê como objeto. Porém, podemos utilizar o `instance_double`, que cria um dublê utilizando uma instância de um objeto como base.

Para fazermos o isso, criaremos um novo contexto e, na definição do nosso objeto, utilizamos `instance_double`.

```
context 'Pokemon' do

  let(:objeto) do
    instance_double(Pokemon)
  end

  it 'retorna um paragrafo por chave' do
```

```
    expect(card_presenter.show).
      to eq(%{<p>nome: Mauro</p><p>idade: 24</p>})
  end
end
```

Ao rodarmos o nosso teste, ele quebra, dado que o nosso model `Pokemon` não implementa o método `#to_presenter`. Para fazermos o nosso teste passar, simplesmente criamos o método `#to_presenter` vazio.

```
class Pokemon < ActiveRecord::Base
  # ...

  def to_presenter
  end
end
```

Agora o nosso teste passa. Isso porque estamos fazendo um stub no objeto, no nosso bloco `before` em que definimos o nome e idade. Não faz sentido o pokémon ter os meus dados, mas vamos deixar ele assim por enquanto.

Além do `instance_double`, o RSpec também implementa o `object_double`, que nos permite passar uma instância de um objeto.

```
object_double(Pokemon.new(nome: 'Charizard'))
```

O `object_double` é útil para quando precisamos ter um estado definido em nosso objeto, como no exemplo anterior em que passamos o nome.

Dublês em uma linha

Quando nossos dublês devem retornar um simples valor na chamada de um método, podemos fazer o stub direto no dublê sem a necessidade do `allow` e do `and_return`.

Vamos alterar o stub do nosso pokémon para retornar um hash com o nome do pokémon definido. Para isso, passamos um hash com o valor que desejamos definir, no nosso caso um hash com a chave `nome`.

```
context 'Pokemon' do
```

```
  let(:objeto) do
    instance_double(Pokemon, to_presenter: {nome: 'Charizard'})
  end

  it 'retorna um paragrafo por chave' do
    expect(card_presenter.show).to eq(%{<p>nome: Charizard</p>})
  end
end
```

Vale lembrar que quando tivermos o `Pokemon#to_presenter` é recomendado removermos o uso do dublê e do stub e usar uma instância do `Pokemon`, isso porque agora o nosso colaborador já foi criado.

Resumindo os dublês

Vamos agora ao resumo de dublês.

- Utilize quando o seu colaborador ainda não foi implementado;
- Utilize para descobrir sua interface. No começo ditamos que iríamos criar o `#to_presenter`, no entanto ele poderia ter saído do nosso TDD simplesmente passando o colaborador e, em seguida, pensaríamos em um nome para o método;
- Utilize sempre que possível os super dublês como o `instance_double` e o `object_double`, dado que assim garantimos que o método pelo menos foi criado no colaborador;
- Remova o dublê trocando-o por uma instância do colaborador, quando o colaborador for criado e implementar os métodos necessários.

7.3 Expectativas em mensagens

Já fizemos a nossa classe `CriadorPokemon` que, como o seu nome diz, é responsável por criar os nossos pokémons. Vamos agora criar um esboço do nosso `AtualizadorPokemon`. Para isso, criamos um `initialize` que recebe um pokémon e um método `update!`, responsável por atualizar o

pokémon. Não entraremos em detalhe de implementação dessa classe, mas o que método `update!` deve fazer é:

- Acessar a API em http://pokeapi.co/api/v1/pokemon/6 em que 6 é o id nacional do pokémon que for passado para o objeto na sua criação;
- Pegar todas as informações do pokémon e atualizar o que foi passado como parâmetro no initialize.

Fica o exercício de implementar esta classe. Para simular a lentidão do acesso a rede colocaremos apenas um `sleep` para o método `update!` ser lento e ele durar 10 segundos para realizar a operação.

```
class AtualizadorPokemon

  def initialize(pokemon)
    @pokemon = pokemon
  end

  def update!
    sleep 10
  end

  private

    attr_reader :pokemon
end
```

Com isso temos a nossa classe `AtualizadorPokemon`. Queremos que seja possível o usuário simplesmente clicar em um botão e atualizar o pokémon. Como estamos em uma app rails, vamos criar uma action `pokemons#update`. Vamos começar o nosso teste desta action. Para isso, criamos um contexto e simplesmente chamamos a nossa action passando um pokémon já criado.

```
describe "PUT 'update'" do

    let!(:pokemon) do
```

```
      Pokemon.create!
    end

    it 'atualiza o Pokemon' do
      put :update, id: pokemon
    end
  end
end
```

Ainda não fizemos nenhuma asserção, ou seja, não testamos nada. Como o que é lento no `AtualizadorPokemon` é o acesso à rede, vimos que podemos utilizar o VCR 2.6. Mas vamos diferenciar um pouco e utilizar as *message expectations*.

Para isso, utilizamos o método `expect`, em que passamos a nossa classe `AtualizadorPokemon`, e o `receive`, que diz qual método ela deve receber, no caso o `new`.

```
it 'atualiza o Pokemon' do
  expect(AtualizadorPokemon).to receive(:new)
  put :update, id: pokemon
end
```

Rodamos o nosso teste e ele quebra. Para o fazermos passar, chamamos o nosso `AtualizadorPokemon` como é esperado no nosso teste.

```
class PokemonsController < ApplicationController
  def update
    AtualizadorPokemon.new
    redirect_to pokemons_path
  end
end
```

Rodamos novamente, e nosso teste passa. No entanto, testamos apenas que ele instanciou o `AtualizadorPokemon`. Não testamos ainda que ele recebeu o objeto correto, nem que chamou o método `update!`.

Vamos continuar e agora testar que ele recebe o objeto correto, o nosso pokémon que está no `let`, que é passado para o controller. Para isso utilizamos o método `with`, que nos dá possibilidade de definir os parâmetros esperados.

```
it 'atualiza o Pokemon' do
  expect(AtualizadorPokemon).to receive(:new).with(pokemon)
  put :update, id: pokemon
end
```

Rodamos nosso teste e, no momento, ele está quebrado. Vamos ao controller para arrumar. Para isso, passamos a nossa instância de pokémon para o `AtualizadorPokemon`.

```
def update
  pokemon = Pokemon.find(params[:id])
  AtualizadorPokemon.new(pokemon)
  redirect_to pokemons_path
end
```

Nossos testes agora passam e estamos cada vez mais próximos de finalizá-los. Falta agora testarmos que o `AtualizadorPokemon` recebeu o método `update!`.

Para testarmos que `AtualizadorPokemon` recebeu um outro método, teremos que utilizar os nossos dublês. Primeiro, definimos um dublê do `AtualizadorPokemon` e, em seguida, utilizamos o `and_return` ao finalizar a nossa chamada do `new`, garantindo que o retorno seja o nosso dublê.

```
it 'atualiza o Pokemon' do
  atualizador_pokemon = double(AtualizadorPokemon)
  expect(AtualizadorPokemon).to receive(:new).with(pokemon)
    .and_return(atualizador_pokemon)
  put :update, id: pokemon
end
```

Agora temos o controle do retorno do `new`, que é o nosso dublê. Fazemos mais uma expectativa, agora no nosso dublê, verificando se ele recebeu o método correto, `update!`.

```
it 'atualiza o Pokemon' do
  atualizador_pokemon = double(AtualizadorPokemon)
  expect(AtualizadorPokemon).to receive(:new).with(pokemon)
    .and_return(atualizador_pokemon)
  expect(atualizador_pokemon).to receive(:update!)
```

```
  put :update, id: pokemon
end
```

Nosso teste, como esperado, está quebrado. Para fazê-lo passar, chamamos o método `update!` no nosso `AtualizadorPokemon`.

```
def update
  pokemon = Pokemon.find(params[:id])
  AtualizadorPokemon.new(pokemon).update!
  redirect_to pokemons_path
end
```

Agora sim nosso teste está completo. Estamos testando que a classe é instanciada, que recebe os parâmetros corretos e chama o método correto. A cereja no topo do bolo: vamos trocar o nosso `double` por `instance_double` para garantir que a classe de que estamos fazendo asserção aqui realmente implementa o método que estamos testando.

```
it 'atualiza o Pokemon' do
  atualizador_pokemon = instance_double(AtualizadorPokemon)
  expect(AtualizadorPokemon).to receive(:new).with(pokemon)
    .and_return(atualizador_pokemon)
  expect(atualizador_pokemon).to receive(:update!)
  put :update, id: pokemon
end
```

Perceba que, quando estamos testando com *message expectations*, nossa estrutura de testes muda: em vez do padrão *setup*, exercício, verificação e *teardown*, temos o *setup*, verificação, exercício e *teardown*. Tenha isso em mente para quando realizar testes que fazem expectativas em mensagens.

Resumindo as message expectations

Vamos agora ao resumo das *message expectations*.

- Utilize para verificar a chamada correta de um colaborador;
- Além de testar a chamada ao método, teste seus argumentos;
- Teste unitariamente o seu colaborador.

7.4 Matchers de argumentos

Nos nossos exemplos anteriores, utilizamos o método `with` que, além de receber valores explícitos, ele aceita alguns matchers. Vamos dar uma olhada neles.

Quando o que importa é a presença

Temos um colaborador do nosso sistema, o `FacebookPost`, que faz uma postagem no facebook utilizando o método `criar`. Ele já possui uma mensagem padrão, então no nosso teste queremos apenas que o colaborador seja executado sem nenhum parâmetro. Para isso, utilizamos o matcher `no_args`.

```
expect(FacebookPost).to receive(:criar).with(no_args)
```

Em outro contexto, queremos que ele tenha recebido uma mensagem qualquer como parâmetro e para isso utilizamos o matcher `anything`.

```
expect(FacebookPost).to receive(:criar).with(anything)
```

E para quando o parâmetro pode estar presente ou não, utilizamos o `any_args`.

```
expect(FacebookPost).to receive(:criar).with(any_args)
```

Assim nosso teste passa se houver parâmetro ou não.

Com expressão regular

No `FacebookPost`, temos que garantir que sempre tenhamos mencionado o termo "pokémon" na mensagem. Para isso, utilizaremos uma expressão regular.

```
expect(FacebookPost).to receive(:criar).with(/pokémon/)
```

Como pode ver, a expressão regular é escrita do mesmo modo que fazemos em Ruby sem a necessidade de um matcher específico.

Quando o paramêtro é um hash

No `AtualizadorPokemon`, esperamos que o nosso pokémon atualize o seu ataque. Então, podemos testar que ele recebe o `update_attributes` com o seu novo ataque. Para não passarmos o hash completo, utilizamos o `hash_including`, que nos permite definir apenas uma parte do hash.

```
expect(pokemon).to receive(:update_attributes)
                   .with(hash_including(ataque: 12))
```

No entanto, não queremos que ele mude o nome do nosso pokémon. Para tal, utilizamos o `hash_not_including`, que nos permite definir a parte do hash que não queremos presente.

```
expect(pokemon).to receive(:update_attributes)
                   .with(hash_not_including(:nome))
```

Ambos os matchers nos permitem definir apenas a chave ou chave e valor, o que nos dá bastante versatilidade.

Quando o que nos importa é a classe

Utilizando nosso `CardPresenter`, estamos esperando que na exibição de pokémons ele receba apenas uma instância de pokémon, não importa qual pokémon. Para escrever este teste utilizamos o matcher `instance_of`.

```
expect(subject).to receive(:new).with(instance_of(Pokemon))
```

Quando não nos importa o objeto exato, mas sim que este seja apenas uma instância de uma dada classe, utilizamos o matcher `instance_of`.

No entanto, na exibição do treinador pokémon utilizamos o `CardPresenter` para exibir, além dos pokémon, o card dos usuários. Então, temos que verificar se o parâmetro recebido é um objeto `ActiveRecord::Base`, e para isso utilizamos o `kind_of`.

```
expect(subject).to receive(:new).
  with(kind_of(ActiveRecord::Base))
```

Agora nosso objeto pode não ser mais um que herda de `ActiveRecord::Base`, então temos que garantir que `CardPresenter` receba um objeto que implemente o método `to_presenter`, não nos importa de qual classe ele seja. Para isso utilizamos o `duck_type`.

```
expect(subject).to receive(:new).with(duck_type(:to_presenter))
```

Além dos matchers que vimos, temos ainda o `boolean`, que garante que o parâmetro é `true` ou `false`. E o `array_including`, que funciona como o `hash_including` recebendo os valores de um array.

Outra funcionalidade legal no uso do `with` é que é possível passarmos mais de um argumento, assim podemos passar matchers junto de valores explícitos como, por exemplo, `with(anything, 12, duck_type(:save))`.

7.5 Um pouco mais sobre stubs, dublês e message expectations

Vamos agora a algumas dicas para quando estamos utilizando os stubs, dublês e message expectations.

Encadeamento de métodos

Os escopos do Active Record nos fornecem uma interface fluente, de modo que podemos fazer `Pokemon.aprovados.recem_criados` com o que aprendemos até aqui. Se precisássemos fazer um stub do `recem_criados`, nesta chamada teríamos que definir antes o retorno de aprovados com um dublê, assim como fizemos no nosso exemplo de expectativa de mensagens 7.3.

Para quando temos que fazer stub de uma chamada a métodos consecutivos, podemos utilizar o `receive_message_chain`. Para isso simplesmente passamos os nomes dos métodos dos quais estamos fazendo o stub.

```
allow(Pokemon).
  to receive_message_chain(:aprovados, :recem_criados).
  and_return([])
```

Como pode ver, esta chamada a métodos encadeados não leva em consideração os parâmetros. Se for importante no seu teste que os parâmetros sejam corretos, utilize o mesmo conceito que fizemos na expectativa de mensagens com o uso de um dublê.

Além de funcionar com o `allow` para criarmos um stub, funciona com o `expect` para fazermos uma asserção.

Quando as coisas dão errado

Em uma API temos uma tela de histórico, e para montá-lo é utilizado o *User Agent* enviado no header. A API continua funcionando sem o header; ele é necessário apenas para vermos estatísticas de uso. Vamos fazer um teste para enviar o User Agent.

Vamos começar com o nosso teste fazendo uma expectativa de que o `Net::HTTP` recebe os parâmetros corretos no header.

```
it 'envia o user agent' do
  expect(Net::HTTP).to receive(:get).with(anything,
      { 'User-Agent' => 'RSpec' })
  acessa_api
end
```

Ao rodarmos nosso teste, recebemos o seguinte erro: `undefined method 'strip'for nil:NilClass`. Nosso método atual que acessa a API possui o seguinte código.

```
resposta = Net::HTTP.get(endpoint, { 'User-Agent' => 'RSpec' })
resposta.strip
```

Como pode ver, depois de pegarmos a resposta, estamos utilizando o `strip` para remover espaços indesejáveis.

O nosso erro `undefined method 'strip'for nil:NilClass` ocorre pois, quando estamos utilizando stub ou fazendo expectativa nas mensagens, o método nunca é chamado, e ele sempre retorna `nil` e este não implementa o método `strip`.

No nosso exemplo podemos resolver simplesmente retornando uma string vazia.

```
expect(Net::HTTP).to receive(:get).with(anything,
    { 'User-Agent' => 'RSpec' }).and_return('')
```

Isso resolve o nosso problema, mas além do `strip`, a nossa resposta pode estar chamando diversos métodos e alguns podem não ser implementados pela string.

Para resolver podemos retornar um dublê para a nossa resposta.

```
resposta = double('resposta HTTP')
expect(Net::HTTP).to receive(:get).with(anything,
  { 'User-Agent' => 'RSpec' }).and_return(resposta)
```

No entanto, agora estamos recebendo outro erro: `Double "resposta HTTP"received unexpected message :strip with (no args)`, ou seja, o nosso dublê não implementa o `strip`. Podemos fazer o stub deste e de N outros métodos, porém seria um processo muito chato e deixaria o setup do nosso teste bem complexo. Neste teste, não nos importamos com as demais chamadas dos métodos em `resposta`, estes são testados em outros testes. O único objetivo que temos aqui é testar se o header foi enviado corretamente. Então podemos dizer que o nosso dublê é um *null object*, isto é, ele responde a qualquer método.

```
resposta = double('resposta HTTP').as_null_object
expect(Net::HTTP).to receive(:get).with(anything,
    { 'User-Agent' => 'RSpec' }).and_return(resposta)
```

Agora nossos testes passam. Para finalizar, podemos utilizar ainda o `and_call_original`, que realiza a chamada do método retornando o seu valor. Assim podemos remover o uso do dublê.

```
expect(Net::HTTP).to receive(:get).with(anything,
    { 'User-Agent' => 'RSpec' }).and_call_original
```

Vale lembrar que agora estamos fazendo a requisição HTTP, então é bom utilizarmos o VCR neste caso.

A dica é: inicie com um simples valor como retorno, e caso encontre problemas utilize do dublê como *null object*; caso o método testado não seja custoso de ser executado, utilize o `and_call_original`.

7.6 Mockar ou não mockar?

Como podemos ver, o uso de *message expectations*, dublês e stub aumenta o nosso acoplamento, dado que se a assinatura de um método mudar, por exemplo, teremos que alterar os testes para este agora refletir esta nova chamada.

Outro ponto que temos que levar em consideração ao utilizarmos estes recursos é quando está complicado de fazer o nosso cenário, como quando estamos criando diversos dublês e fazendo stub de diversos métodos. Temos que especificar apenas o necessário para o nosso exemplo passar e, se ainda assim isso for muita coisa, provavelmente nossa classe/método está fazendo coisa demais e clamando por um refactoring.

Lembre-se de que é mais complicado para um iniciante entender testes utilizando de mocks do que a abordagem clássica, então pense no seu time antes de começar a utilizar esta abordagem. Veja se todos estão confortáveis e utilize-a apenas quando for necessária.

Temos estas duas escolas: a clássica e a de mocks. Pessoalmente, eu utilizo a abordagem clássica com um pouquinho de mock. Utilizo mocks basicamente quando algum colaborador meu, criado por mim, necessita ser executado, como, no nosso exemplo, o `AtualizadorPokemon` em *message expectations*. Isso porque a abordagem clássica me dá maior confiança.

7.7 Conclusão

- Conhecemos o stub para forjar o retorno de nossos métodos;
- Vimos que devemos forjar apenas nos nossos colaboradores;
- Utilizamos os dublês quando ainda não tínhamos o nosso colaborador criado;
- Conhecemos os super dublês que nos dão uma maior confiança;
- Fizemos asserção na chamada de métodos utilizando as *message expectations*;
- Conhecemos diversos matchers de argumentos;

- Utilizamos o encadeamento de métodos;
- Vimos que às vezes os mocks podem sair do trilho;
- Discutimos um pouco sobre esta abordagem ao se realizar os testes.

Fique por aí, que no próximo capítulo veremos que temos modos de fazer debug além do `puts` e como melhorar a nossa experiência enquanto estamos no nosso console. É um capítulo bem curtinho e mais tranquilo. Ele não é 100% relacionado com testes, mas quem nunca escreveu um `puts` pra fazer debug nos testes?

Capítulo 8

Não debugamos com puts, certo?

Quantas vezes já nos deparamos com algo assim no meio de nosso código:

```
puts 'oi'
```

Pode ser um oi, um palavrão, qualquer palavra, ou ainda uma variável do nosso código. Normalmente fazemos isso quando queremos saber se nosso código passou por um determinado método ou condição, e ainda quando queremos saber o valor de uma variável em tal momento de execução. Mas será que não temos um modo melhor de fazer isso?

Como você já pode, perceber estamos falando de debug. Dado que passamos 60% do nosso tempo fazendo debug, é bom darmos uma atenção a isso e termos uma forma mais professional do que o `puts`. Pessoalmente, quando estou testando algo que depende de uma gem que nunca usei ou uma api que desconheço, faço meu teste enquanto vou debugando o funcionamento de tal api.

8.1 Por um melhor console

Antes de falarmos sobre debug, vamos antes dar um upgrade em como o nosso console exibe as informações, o que vai nos ajudar muito no debug.

No Ruby, é comum utilizarmos bastante o nosso console, seja para iniciarmos o Rails, rodar testes, excutarmos *rake tasks* etc. Também utilizamos o console do próprio Rails para darmos uma olhada nos nossos objetos com um `rails c`. No entanto, vamos dar uma olhada em como é exibido um objeto para nós no console do Rails. Para isso, vamos simplesmente dar uma olhada em uma instância de um pokémon qualquer.

```
1.1.0 :001 > pokemon
 => #<Pokemon id: 1, id_nacional: nil, nome: nil,
 ataque: nil, defesa: nil, created_at: "2015-04-04 22:11:22",
 updated_at: "2015-04-04 21:11:23">
```

Como se pode ver, a exibição é bem confusa: são exibidos todos os campos em apenas uma linha (no nosso exemplo, a quebra de linha é para se adequar ao livro) e não temos cores para nos ajudar. Se já está confuso com um simples objeto, imagine em um modelo com mais campos!

O Awesome Print

O Awesome Print é uma gem que nos ajuda na tarefa de exibir nossos objetos Ruby com melhor formatação. Para instalar, simplesmente adicionamos ao nosso gemfile o `awesome_print` e executamos `$ bundle install`.

```
group :development, :test do
  # ...
  gem 'awesome_print'
end
```

Para finalizar e já termos a exibição do Awesome Print por padrão no nosso irb, criamos um arquivo `.irbrc` no nosso home com o seguinte conteúdo:

```
require "awesome_print"
AwesomePrint.irb!
```

Vamos voltar para o nosso console agora e exibir o mesmo objeto.

```
1.1.0 :001 > pokemon
#<Pokemon:0x000001049cf8d8> {
              :id => 1,
    :id_nacional => nil,
           :nome => nil,
         :ataque => nil,
         :defesa => nil,
     :created_at => Mon, 04 Apr 2015 22:11:22 BRT
     -03:00, :updated_at => Mon, 04 Apr 2015 22:11:23 BRT -03:00
}
```

Como se pode ver, agora está bem mais claro o valor de cada um dos atributos do nosso modelo, além de termos cores que facilitam e muito o nosso entendimento.

8.2 Conhecendo o Pry

O Pry não é apenas um debug, mas sim um debug bombado. Por hora, vamos começar apenas o instalando. Para isso, adicionamos o `pry-rails` no nosso `Gemfile`, que nos fornece os superpoderes do pry em um console rails e executamos `$ bundle install`.

```
group :development, :test do
  # ...
  gem 'pry-rails'
end
```

Não queremos perder a nossa formatação do Awesome Print e, para isso, criamos um arquivo `.pryrc` no nosso diretório home e adicionamos as seguintes linhas:

```
require "awesome_print"
AwesomePrint.pry!
```

Assim, temos o Awesome Print integrado direto com o pry. Mas vamos agora dar uma olhada neste pry.

Documentação ao seu alcance

Quando criamos o nosso `CriadorPokemon`, utilizamos o `Net::HTTP`, mas como podemos saber quais são os métodos, qual é assinatura destes métodos etc.? Podemos conseguir esta informação acessando a API do Ruby pelo browser, mas se já conseguíssemos todas estas informações direto pelo console? Para isso, o Pry nos oferece o comando `ls` que lista todos os métodos de uma classe ou objeto.

```
ls Net::HTTP
```

Como resposta, teremos os métodos, atributos e demais informações da classe. Veja um trecho da resposta.

```
#<Class:Net::HTTP>#methods:
  Proxy         get_print
  default_port  get_response
  get           http_default_port
```

Como se pode ver, a nossa classe `Net::HTTP` implementa o método `get`, que é bem provavel ser o que precisamos quando queremos fazer uma requisição `get`. Então agora vamos ler a sua documentação para saber como utilizá-lo, isso ainda dentro do console, por meio do comando `?`

```
? Net::HTTP#get
```

Que nos fornece como resposta a mesma documentação que teríamos ao acessar a API do Ruby. Vamos a um pequeno trecho:

```
From: /path/ruby-2.1.0/lib/ruby/2.1.0/net/http.rb @ line 1089:
Owner: Net::HTTP
Visibility: public
Signature: get(path, initheader=?, dest=?, &block)
Number of lines: 37

Retrieves data from path on the connected-to host which may be an
absolute path String or a URI to extract the path from.
```

O comando `?` também pode ser acessado através do `show-doc`. Como se pode ver, é muito util, pois não precisamos sair do nosso terminal para acessar a documentação.

Podemos ainda ir além e ver a implementação de um dado método ou classe. Para isso, utilizamos o comando `$`.

```
$ Net::HTTP#get
```

Isso nos retorna à implementação do método `get` da classe `Net::HTTP`. Vamos a um trecho.

```
From: /path/ruby-2.1.0/lib/ruby/2.1.0/net/http.rb @ line 1122:
Owner: Net::HTTP
Visibility: public
Number of lines: 8

def get(path, initheader = {}, dest = nil, &block)
# :yield: +body_segment+
```

O comando `$` também pode ser acessado através do comando `show-source`.

Conseguimos acessar bastante documentação diretamente do nosso console sem a necessidade de sairmos do terminal, o que facilita em muito o nosso desenvolvimento.

Para finalizarmos a nossa sessão do Pry podemos utilizar u m `< Ctrl> + d` ou com o comando `exit`.

Utilizando o Pry em tempo de execução

Agora vamos ver o que para mim é a melhor funcionalidade do Pry, que a possibilidade de utilizar o Pry em tempo de execução. Vamos alterar o `CriadorPokemon` e chamar o Pry com o `binding.pry`.

```
def cria_info
  binding.pry
  resposta = Net::HTTP.get(endpoint)
  @info = JSON.parse(resposta)
end
```

Agora vamos rodar o nosso teste. Deparamo-nos com a seguinte mensagem no console:

```
From: /path/criador_pokemon.rb @ line 21
CriadorPokemon#cria_info:

    20: def cria_info
 => 21:   binding.pry
    22:   resposta = Net::HTTP.get(endpoint)
    23:   @info = JSON.parse(resposta)
    24: end

[1] pry(#<CriadorPokemon>)>
```

Perceba que quando o teste atinge o método `cria_info` ele será parado e nos levará ao console do Pry. Ok, mas e aí? Temos um console Ruby então podemos rodar diversos comandos que desejamos, inclusive os do próprio Pry que vimos anteriormente.

Observe que no console é exibido o contexto em que estamos. No nosso caso, é o `CriadorPokemon`. Se utilizamos o `ls` e os demais comandos que o Pry nos fornece, veremos os métodos e variáveis do objeto atual, ou seja do nosso `CriadorPokemon`.

Como estamos em um ambiente de debug, vamos rodar a linha 22 e 23, podemos copiar e colar... é isso não é legal né? Então, para não precisar fazer isso, o Pry nos fornece o comando `play -l`, que aceita como parêmetro um número ou *range*. Vamos passar um range de 22 a 23.

```
play -l 22..23
```

Assim executamos as 2 linhas uma seguida da outra. Como resultado será exibido o hash gerado pelo `JSON.parse`. Agora eu quero ver o valor da variável... é qual o nome da variável? Para nos ajudar com isso o Pry nos fornece o método `whereami` que nos retorna para o ponto onde definimos o nosso `binding.pry`.

Agora que estamos de volta e descobrimos que o nome da variável que armazenamos o valor do `JSON.parse` foi `@info`, vamos dar uma olhada nela. Podemos simplesmente chama-lá diretamente com `@info`, mas vamos fazer diferente: vamos trocar o nosso contexto. Para isso, utilizamos o comando `cd` seguido da variável que entraremos, no nosso caso `@info`.

```
cd @info
```

Se observarmos no console, veremos que o nosso contexto mudou agora para `#<Hash>):1` em vez do `#<CriadorPokemon>`. E como você já deve imaginar, agora podemos acessar os comandos do Pry no nosso contexto assim como qualquer método do Hash, como por exemplo `keys`, sem a necessidade de chamar o objeto, afinal agora estamos no contexto do próprio objeto.

Para voltarmos para o nosso contexto pai, simplesmente executamos o comando `cd ..`.

A ultima saída que tivemos no console, foi a chamada do método `keys`. Podemos acessar a última saída através da variável _, podemos passá-la como parâmetro e executar qualquer método, por exemplo `_[0]`.

Até aqui já digitamos diversos comandos e tivemos várias saídas. Todas essas entradas e saídas podem ser acessadas nas váriaveis `_in_` e `_out_`, respectivamente. Além disso, podemos utilizar do comando `<Ctrl> + r` para realizar uma busca enquanto começamos digitar um comando.

Eu não mencionei aqui, mas quando estiver realizando os seus testes verá que o Pry nos fornece bastantes cores enquanto estamos utilizando seus comandos.

Eu pessoalmente utilizo muito autocomplete. Se você também curte, no Pry temos autocomplete simplesmente digitando `<tab>`.

Como se pode ver, o uso do Pry ajuda muito no nosso processo de desenvolvimento. Particularmente, eu o utilizo até quando escrevo meus testes, pois consigo ter um ambiente controlado, com apenas o cenário do meu teste, diferente de eu abrir um console que terá muito mais dados e eles podem não estar com os valores definidos de que eu preciso.

O ecossistema

Além do `pry-rails` que utilizamos aqui, o Pry tem todo um ecossistema de gems que ajudam nesse processo de debug, vamos a alguma delas.

- **pry-rescue:** abre o Pry sempre que uma excessão é lançada, https://github.com/ConradIrwin/pry-rescue.

- **pry-stack_explorer:** permite navegar pelo *stack*, https://github.com/pry/pry-stack_explorer.
- **pry-debugger:** adiciona mais comandos de debug ao Pry e a possibilidade de adicionar breakpoints, https://github.com/nixme/pry-debugger.
- **pry-plus:** coleção de ferramentas para aumentar os poderes do Pry, https://github.com/rking/pry-plus.
- **jazz_hands:** outra coleção de ferramentas que inclui o pry-rails e Awesome Print, https://github.com/nixme/jazz_hands.

Antes de adicionar estas demais gems, experimente primeiro apenas com o Pry e ao Awesome Print e, depois, vá adicionando o que achar legal ao seu repertório, assim terá mais conhecimento sobre cada uma das ferramentas.

8.3 Conclusão

- Vimos que podemos melhorar o nosso console utilizando o Awesome Print;
- Acessamos a documentação direto do console com o uso do Pry;
- Utilizamos o Pry em tempo de execução;
- Fomos apresentados ao ecossistema do Pry.

Capítulo 9

Conclusão

Espero que tenha aproveitado a nossa aventura, aprendido formas melhores de se escrever testes e conhecido novas ferramentas e técnicas para ajudar no processo de TDD.

Enquanto estou lendo um livro costumo traçar o paralelo de que cada capítulo é como se fosse uma fase do Megaman e, quando finalizo um capítulo, derrotei o chefe da fase e ganhei seus poderes e agora posso usar este novo poder como bem entender. Espero que tenha derrotado todos os chefes e ganhando novos poderes você também.

Gostaria de ouvir seu feedback, então não deixe de me contatar para questões, comentários, sugestões e para informar qualquer erro que encontrar. Meu e-mail é mauro@helabs.com.br.

Se gostou do conteúdo do livro, eu o convido para visitar o meu blog sobre desenvolvimento de software em http://groselhas.maurogeorge.com.br.

Obrigado por chegar até aqui e *happy hacking*.

Referências Bibliográficas

[1] Peter Cooper. Dhh offended by rspec, says test::unit is just great. http://www.rubyinside.com/dhh-offended-by-rspec-debate-4610.html, 2011.

[2] Joe Ferris. Let's not. http://robots.thoughtbot.com/lets-not, 2012.

[3] David Heinemeier Hansson. Testing like the tsa. http://signalvnoise.com/posts/3159-testing-like-the-tsa, 2012.

[4] Steve Klabnik. Why i don't like factory_girl. http://blog.steveklabnik.com/posts/2012-07-14-why-i-don-t-like-factory_girl, 2012.

[5] Robert C. Martin. Am i suggesting 100 https://twitter.com/unclebobmartin/status/55954057327677440, 2011.

[6] Andrea Reginato. Better specs rspec guidelines with ruby . http://betterspecs.org/, 2012.

[7] Josh Steiner. How we test rails applications. http://robots.thoughtbot.com/how-we-test-rails-applications, 2014.