CAPÍTULO 16

Apêndice: Testes

"Ninguém testa a profundidade de um rio com os dois pés."
— Provérbio Africano

16.1 - O PORQUÊ DOS TESTES?

Testes de Unidade são classes que o programador desenvolve para se certificar que partes do seu sistema estão funcionando corretamente.

Eles podem testar validações, processamento, domínios etc, mas lembre-se que um teste unitário deve testar *somente um pedaço de código* (de onde veio o nome *unitário*).

Criar esse tipo de testes é uma das partes mais importantes do desenvolvimento de uma aplicação pois possibilita a verificação real de todas as partes do programa automaticamente.

> **Extreme Programming (XP)**
>
> Extreme Programming é um conjunto de práticas de programação que visam a simplicidade, praticidade, qualidade e flexibilidade de seu sistema. Os testes de unidade fazem parte dessa metodologia de programação.

O Ruby já possui classes que nos auxiliam no desenvolvimento destes testes.

16.2 - TEST::UNIT

`Test::Unit` é a biblioteca usada para escrever suas classes de teste.

Ao escrever testes em Ruby utilizando esse framework, você deve herdar a classe `TestCase` que provê a funcionalidade necessária para fazer os testes.

```
require 'test/unit'

class PessoaTest < Test::Unit::TestCase
  # ...
end
```

Ao herdar `Test`*`Unit`*`TestCase`, você ganha alguns métodos que irão auxiliar os seus testes:

- `assert(boolean, msg=nil)`
- `assert_equal(esperado, atual, msg=nil)`
- `assert_not_equal(esperado, atual, msg=nil)`
- `assert_in_delta(esperado, atual, delta, msg=nil)`
- `assert_instance_of(classe, objeto, msg=nil)`
- `assert_kind_of(classe, objeto, msg=nil)`
- `assert_match(regex, texto, msg=nil)`
- `assert_no_match(regex, texto, msg=nil)`
- `assert_nil(objeto, msg=nil)`
- `assert_not_nil(objeto, msg=nil)`
- `assert_respond_to(objeto, metodo, msg=nil)`
- `assert_same(esperado, atual, msg=nil)`
- `assert_not_same(esperado, atual, msg=nil)`

O método `assert` simples recebe como parâmetro qualquer expressão que devolva um valor booleano e todos os métodos `assert` recebem opcionalmente como último argumento uma mensagem que será exibida caso a asserção falhe.

Mais detalhes e outros métodos `assert` podem ser encontrados na documentação do módulo `Test`*`TestUnit`*`Assertions`, na documentação da biblioteca core da linguagem Ruby (http://ruby-doc.org/core/).

Os testes podem ser executados em linha de comando, bastando chamar `ruby`

`o_que_eu_quero_testar.rb`. O resultado é um **"."** para os testes que passarem, **"E"** para erros em tempo de execução e **"F"** para testes que falharem.

Também é possível executar todos os testes com algumas tasks do rake:

```
# roda todos os testes de unidade, de integração e funcionais
rake test

# roda todos os testes da pasta test/unit
rake test:units

# roda todos os testes da pasta test/functional
rake test:functionals

# roda todos os testes da pasta test/integration
rake test:integration

# roda todos os testes de plugins, na pasta vendor/plugins
rake test:plugins
```

Existem ainda outras tarefas disponíveis para o rake. Sempre podemos consultá-las com `rake -T`, no diretório do projeto.

Podemos criar uma classe de teste que só possua um único "assert true", no diretório test/unit/.

```
class MeuTeste < Test::Unit::TestCase
  def test_truth
    assert true
  end
end
```

Ao escrever testes de unidade em projetos Ruby On Rails, ao invés de herdar diretamente de `Test`*`Unit`*`TestCase`, temos a opção de herdar da classe fornecida pelo ActiveSupport do Rails:

```
require 'test_helper'

class RestauranteTest < ActiveSupport::TestCase
  def test_anything
    assert true
  end
end
```

Além disso, todos os testes em projetos Rails devem carregar o arquivo **test_helper.rb**, disponível em qualquer projeto gerado pelo Rails. As coisas comuns a todos os testes, como método utilitários e configurações, ficam neste arquivo.

A vantagem de herdar de `ActiveSupport::TestCase` ao invés da original é que o

Rails provê diversas funcionalidades extras aos testes, como fixtures e métodos `assert` extras. Alguns dos asserts extras:

- `assert_difference`
- `assert_no_difference`
- `assert_valid(record)` – disponível em testes de unidade
- `assert_redirected_to(path)` – para testes de controladores
- `assert_template(esperado)` – também para controladores
- entre outros

16.3 – Exercícios – Teste do modelo

1. Vamos testar nosso modelo restaurante. Para isso precisamos utilizar o banco de dados específico para testes.

```
rake db:create:all
rake db:migrate RAILS_ENV=test
```

2. O Rails já tem suporte inicial para testes automatizados. Nossa aplicação já possui arquivos importantes para nossos testes. Abra o arquivo **test/fixtures/restaurantes.yml**. Esse arquivo simula os dados de um restaurante. Crie os dois restaurantes abaixo. **Importante:** Cuidado com a identação!

```
fasano:
  nome: Fasano
  endereco: Rua Vergueiro

fogo_de_chao:
  nome: Fogo de Chao
  endereco: Avenida dos Bandeirantes
```

3. O arquivo de teste do modelo está em **test/unit/restaurante_test.rb**.

```
require 'test_helper'

class RestauranteTest < ActiveSupport::TestCase
  fixtures :restaurantes

    def test_restaurante

      restaurante = Restaurante.new(
         :nome => restaurantes(:fasano).nome,
               :endereco => restaurantes(:fasano).endereco,
         :especialidade => restaurantes(:fasano).especialidade)

          msg = "restaurante não foi salvo. "
      + "errors: ${restaurante.errors.inspect}"
      assert restaurante.save, msg

      restaurante_fasano_copia = Restaurante.find(restaurante.id)

      assert_equal restaurante.nome, restaurante_fasano_copia.nome
    end
end
```

4. Para rodar o teste, vá na raiz do projeto pelo terminal e digite:

```
rake test
```

Verifique se tudo está certo:

```
Loaded suite test/unit/restaurante_test
Started
.
Finished in 0.044991 seconds.

1 tests, 4 assertions, 0 failures, 0 errors
```

16.4 – Exercícios – Teste do controller

1. Para testar o controller de restaurantes vamos criar uma nova action chamada **busca**. Essa action direciona para o restaurante buscado caso encontre ou devolve uma mensagem de erro caso contrário. Abra o arquivo **app/controllers/restaurantes_controller.rb** e adicione a action **busca**:

```
def busca
  @restaurante = Restaurante.find_by_nome(params[:nome])
  if @restaurante
    redirect_to :action => 'show', :id => @restaurante.id
  else
    flash[:notice] = 'Restaurante não encontrado.'
    redirect_to :action => 'index'
  end
```

```
end
```

2. Abra o arquivo **test/functional/restaurantes_controller_test.rb**.

```
require 'test/test_helper'

class RestaurantesControllerTest < ActionController::TestCase
  fixtures :restaurantes

    def test_procura_restaurante
      get :busca, :nome => 'Fasano'
      assert_not_nil assigns(:restaurante)
      assert_equal restaurantes(:fasano).nome, assigns(:restaurante).nome
      assert_redirected_to :action => 'show'
    end

   def test_procura_restaurante_nao_encontra
      get :busca, :nome => 'Botequin'
      assert_redirected_to :action => 'index'
      assert_equal 'Restaurante não encontrado.', flash[:notice]
    end
end
```

Verifique se tudo está certo;

```
Loaded suite test/functional/restaurantes_controller_test
Started
..
Finished in 0.206066 seconds.

2 tests, 4 assertions, 0 failures, 0 errors
```

3. Rode o teste no terminal com **rake test**.

16.5 – RSpec

Muito mais do que uma nova forma de criar testes de unidade, RSpec fornece uma forma de criar especificações executáveis do seu código.

No TDD, descrevemos a funcionalidade esperada para nosso código através de testes de unidade. BDD (*Behavior Driven Development*) leva isso ao extremo e diz que nossos testes de unidade devem se tornar especificações executáveis do código. Ao escrever as especificações estaremos pensando no **comportamento esperado** para nosso código.

> **Introdução ao BDD**
>
> Uma ótima descrição sobre o termo pode ser encontrada no site do seu

próprio criador: Dan North.

http://dannorth.net/introducing-bdd/

RSpec fornece uma DSL (*Domain Specific Language*) para criação de especificações executáveis de código. As especificações do RSpec funcionam como exemplos de uso do código, que validam se o código está mesmo fazendo o que deveria e funcionam como documentação.

http://rspec.info

Para instalar o rspec e usar em qualquer programa Ruby, basta instalar o gem:

```
gem install rspec
```

Para usar em aplicações Rails, precisamos instalar mais um gem que dá suporte ao rspec ao Rails. Além disso, precisamos usar o gerador que vem junto desta gem, para adicionar os arquivos necessários nos projetos que forem usar rspec:

```
cd projeto-rails
rails generate rspec:install
```

O último comando também adiciona algumas tasks do rake para executar as specs do projeto, além de criar a estrutura de pastas e adicionar os arquivos necessários.

```
rake spec     # executa todas as specs do projeto
rake -T spec  # para ver as tasks relacionadas ao rspec
```

O rspec-rails também pode ser instalado como plugin, porém hoje é altamente recomendado seu uso como gem. Mais detalhes podem ser encontrados na documentação oficial

http://wiki.github.com/rspec/rspec-rails/

RSpec é compatível com testes feitos para rodar com `Test::Unit`. Desta forma, é possível migrar de forma gradativa. Apesar disso, a sintaxe oferecida pelo RSpec se mostra bem mais interessante, já que segue as ideias do Behavior Driven Development e faz com que os testes se tornem especificações executáveis do código:

```
describe Restaurante, " com nome" do
  it "should have name"
    Restaurante.all.should_not be_empty
    Restaurante.first.should_not be_nil
```

```
    Restaurante.first.name.should == "Fasano"
  end
end
```

A classe de teste vira um **Example Group** (`describe`). Cada método de teste vira um **Example** (`it "should ..."`).

Além disso, os métodos `assert` tradicionais do `Test::Unit` viram uma chamada de `should`. O RSpec adiciona a **todos** os objetos os métodos `should` e `should_not`, que servem para validarmos alguma condição sobre o estado dos nossos objetos de uma forma mais legível e expressiva que com asserts.

Como argumento para o método `should`, devemos passar uma instância de `Matcher` que verifica uma condição particular. O RSpec é extremamente poderoso, pois nos permite escrever nossos próprios Matchers. Apesar disso, já vem com muitos prontos, que costumam ser mais do que suficientes:

- `be_<nome>` para métodos na forma `<nome>?`.

```
# testa: objeto.empty?
objeto.should be_empty

# testa: not objeto.nil?
objeto.should_not be_nil

# testa: objeto.kind_of(Restaurante)
objeto.should be_kind_of(Restaurante)
```

Além de be_<nome>, também podemos usar be_a_<nome> ou be_an_<nome>, aumentando a legibilidade.

- `be_true`, `be_false`, `eql`, `equal`, `exist`, `include`:

```
objeto.should be_true
  objeto.should_not be_false

# testa: objeto.eql?(outro)
  objeto.should eql(outro)

  # testa: objeto.equal?(outro)
objeto.should equal(outro)

  objeto.should exist  # testa: objeto.exist?
  [4,5,3].should include(3)  # testa: [4,5,3].include?(3)
```

- `have_<nome>` para métodos na forma `has_<nome>?`.

```
itens = { :um => 1, :dois => '2' }

# testa: itens.has_key?(:dois)
  itens.should have_key(:dois)
```

```
  # testa: not itens.has_value?(/3/)
  itens.should_not have_value(/3/)
```

- `be_close`, inclui tolerância.

```
conta = 10.0 / 3.0
conta.should be_close(3.3, 0.1) # == 3.3 ~0.1
```

- `have(num).<colecao>`, para testar a quantidade de itens em uma associação.

```
# testa categoria.produtos.size == 15
categoria.should have(15).produtos
```

Um uso especial deste *matcher* é para objetos que já são coleções. Neste caso, podemos usar o nome que quisermos:

```
array = [1,2,3]

  # testa array.size == 3
  array.should have(3).items

# mesma coisa
  array.should have(3).numbers
```

- `have_at_least(num).<colecao>`: mesma coisa que o anterior, porém usa >=.

- `have_at_most(num).<colecao>`: mesma coisa que o anterior, porém usa <=.

- `match`, para expressões regulares.

```
# verifica se começa com F
  texto.should match(/^F/)
```

Este são os principais, mas ainda existem outros. Você pode encontrar a lista de Matchers completa na documentação do módulo `Spec::Matchers`:

Exemplos pendentes

Um exemplo pode estar vazio. Desta forma, o RSpec o indicará como pendente:

```
describe Restaurante do
  it "should have endereco"
end
```

Isto facilita muito o ciclo do BDD, onde escrevemos o teste primeiro, antes do código de verdade. Podemos ir pensando nas funcionalidades que o sistema deve ter e deixá-las pendentes, antes mesmo de escrever o código. Em outras palavras, começamos especificando o que será escrito.

Before e After

Podemos definir algum comportamento comum para ser executado antes ou depois de cada um dos exemplos, como o `setup` e o `teardown` do `Test::Unit`:

```
describe Restaurante do
  before do
    @a_ser_testado = Restaurante.new
  end

  it "should ..."

  after do
    fecha_e_apaga_tudo
  end
end
```

Estes métodos podem ainda receber um argumento dizendo se devem ser executados novamente para cada exemplo (`:each`) ou uma vez só para o grupo todo (`:all`):

```
describe Restaurante do
  before(:all) do
    @a_ser_testado = Restaurante.new
  end

  it "should ..."

  after(:each) do
    fecha_e_apaga_tudo
  end
end
```


16.6 - Cucumber, o novo Story Runner

RSpec funciona muito bem para especificações em níveis próximos ao código,

como as especificações unitárias.

User Stories é uma ferramenta indicada para especificações em níveis mais altos, como funcionalidades de negócio, ou requisitos. Seu uso está sendo bastante difundido pela comunidade Rails. User Stories costumam ter o seguinte formato:

```
In order to <benefício>
As a <interessado>
I want to <funcionalidade>.
```

Cucumber é uma excelente biblioteca escrita em Ruby, que serve para tornar especificações como esta, na forma de *User Stories*, escritas em texto puro, executáveis. Cucumber permite a associação de código Ruby arbitrário, usualmente código de teste com RSpec, a cada um dos passos desta descrição da funcionalidade.

Para instalar tudo o que é necessário:

```
gem install cucumber capybara database_cleaner
```

```
gem 'database_cleaner'
gem 'cucumber-rails'
gem 'cucumber'
gem 'rspec-rails'
gem 'spork'
gem 'launchy'
```

Para projetos rails, é possível usar o *generator* fornecido pelo Cucumber para adicionar os arquivos necessários ao projeto:

```
cd projetorails
rails generate cucumber:install
```

As User Stories são chamadas de **Features** pelo Cucumber. São arquivos de texto puro com a extensão *.feature*. Arquivos com definição de features sempre contém uma descrição da funcionalidade (**Story**) e alguns exemplos (**Scenarios**), na seguinte estrutura:

```
Feature: <nome da story>
  In order to <beneficio>
  As a <interessado>
  I want to <funcionalidade>

  Scenario: <nome do exemplo>
    Given <pré condições>
    And   <mais pré condições>
    When  <ação>
    And   <mais ação>
```

```
    Then  <resultado>
    And   <mais resultado>

  Scenario: <outro exemplo>
    Given ...
    When ...
    Then ...
```

Antigamente, o RSpec incluia sua própria implementação de Story Runner, que hoje está sendo substituída pelo Cucumber. O RSpec Story Runner original utilizava um outro formato para features, mais tradicional, que não dá prioridade ao *Return Of Investment*. O benefício da funcionalidade fica em segundo plano, no final da descrição:

```
Story: transfer from savings to checking account
  As a savings account holder
  I want to transfer money from my savings account to my checking account
  So that I can get cash easily from an ATM

  Scenario: ...
```

O importante para o Cucumber são os exemplos (**Scenarios**) que explicam a funcionalidade. Cada um dos Scenarios contém um conjunto de passos, que podem ser do tipo **Given** (pré-requisitos), **When** (ações), ou **Then** (resultado).

A implementação de cada um dos passos (*steps*) dos *scenarios* devem ficar dentro do diretório **step_definitions/**, na mesma pasta onde se encontram os arquivos *.feature*, texto puro.

O nome destes arquivos que contém a definição de cada um dos passos deve terminar com `_steps.rb`. Cada passo é representado na chamada dos métodos `Given`, `Then` ou `When`, que recebem como argumento uma **String** ou **expressão regular** batendo com o que estiver escrito no arquivo de texto puro (*.feature*).

Tipicamente, os projetos contém um diretório **features/**, com a seguinte estrutura:

```
projeto/
|-- features/
|   |-- minha.feature
|   |-- step_definitions/
|   |   |-- alguns_steps.rb
|   |   `-- outros_steps.rb
|   `-- support/
|       `-- env.rb
```

O arquivo **support/env.rb** é especial do Cucumber e sempre é carregado antes da execução dos testes. Geralmente contém a configuração necessária para os

testes serem executados e código de suporte aos testes, como preparação do Selenium ou Webrat.

Os arquivos com definições dos passos são arquivos Ruby:

```
Given "alguma condição descrita no arquivo texto puro" do
  # código a ser executado para este passo
end

Given /e outra condicao com valor: (.*)/ do |valor|
  # código de teste para esse passo
end

When /alguma acao/
  # ...
end

Then /verifica resultado/
  # ...
end
```

O código de teste para cada passo pode ser qualquer código Ruby. É comum o uso do RSpec para verificar condições (métodos `should`) e **Webrat** ou **Selenium** para controlar testes de aceitação. Mais detalhes sobre estes frameworks para testes de aceitação podem ser vistos no capítulo *"Outros testes e specs"*.

Não é necessário haver um arquivo com definição de passos para cada arquivo de feature texto puro. Isto é até considerado má prática por muitos, já que inibe o reuso para definições de *steps*.

CAPÍTULO ANTERIOR:

Algumas Gems Importantes

PRÓXIMO CAPÍTULO:

Apêndice: Rotas e Rack

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter