thoughtbot

BDD with Shoulda

Tammer Saleh
tammersaleh.com
tsaleh@thoughtbot.com

Expectations

Talk about BDD

Talk about Shoulda

Talk about testing in general

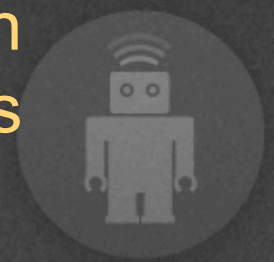
What is BDD

New way of thinking about TDD

Describe behavior instead of writing tests

Write short specs that describe one piece of behavior at a time.

> "It must be stressed that BDD is a **rephrasing of existing good practice**, it is **not a radically new departure**. Its aim is to bring together existing, well-established techniques under a common banner and with a consistent and unambiguous terminology." - behaviour-driven.org



What is Shoulda

Nested contexts and readable test names

Fully compatible with Test::Unit

ActiveRecord & ActionController macros that cover the most common 80%

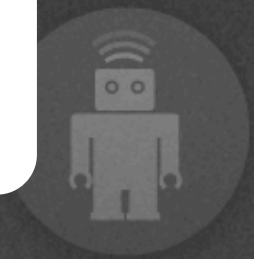
Some RESTful magic for kicks

should statements

```
class QuoteTest < Test::Unit::TestCase
  def setup
    # normal Test::Unit setup stuff here
  end

  def test_should_be_true
    assert true
  end

  should "have a name" do
    assert_respond_to Quote, :name
  end
end
```



Just normal tests

```
define_method "test: Quote should have a name. " do
  assert_respond_to Quote, :name
end
```

contexts

```
class QueueTest < Test::Unit::TestCase
  context "A Queue instance" do
    setup { @queue = Queue.new }

    should "respond to #push" do
      assert_respond_to @queue, :push
    end
  end
end
```

Just normal tests
(still)

```
define_method "test: Quote should have a name. " do
  assert_respond_to Quote, :name
end

define_method "test: A Queue instance should respond to #push. " do
  @queue = Queue.new
  assert_respond_to @queue, :push
end
```

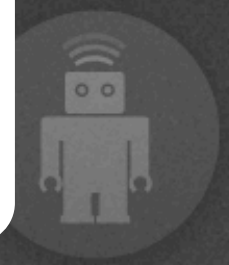
```ruby
class QueueTest < Test::Unit::TestCase
  context "A Queue instance" do
    setup { @queue = Queue.new }

    should "respond to #push" do
      assert_respond_to @queue, :push
    end

    context "with a single element" do
      setup { @queue.push(:something) }

      should "return that element on #pop" do
        assert_equal :something, @queue.pop
      end

      should "have a #size of 1" do
        assert_equal 1, @queue.size
      end
    end
  end
end
```



That's it.

(for the gem)

Macros
(and DRY in general)

Not just to save keystrokes

Faster and easier to read

Reduce programming errors

Distill best practices

ActiveRecord

If it's easy to write in ActiveRecord, it should be just as easy to test.

Covers the most common 80% of the ActiveRecord macros

```ruby
class UserTest < Test::Unit::TestCase
  setup { @user = User.create(...) }

  should_require_attributes :name, :phone_number
  should_require_unique_attributes :name
  should_not_allow_values_for :phone_number,
                              "abcd", "1234"
  should_allow_values_for :phone_number,
                          "(123) 456-7890"

  should_protect_attributes :admin

  should_have_one :profile
  should_have_many :dogs
  should_have_many :messes, :through => :dogs
  should_belong_to :lover
end
```

ActiveRecord

```
test: Person should belong to lover.
test: Person should have many dogs.
test: Person should have many messes through dogs.
test: Person should have one profile.
test: Person should require phone_number to be set.
test: Person should allow phone_number to be set to "(123) 456-7890".
test: Person should not allow phone_number to be set to "1234".
test: Person should not allow phone_number to be set to "abcd".
test: Person should not allow admin to be changed by update.
test: Person should require name to be set.
test: Person should require unique value for name.
```

Controllers

```
class UsersControllerTest < Test::Unit::TestCase
  context "on GET to :show" do
    setup { get :show, :id => 1 }

    should_assign_to :user
    should_respond_with :success
    should_render_template :show
    should_not_set_the_flash

    should "do something else really cool" do
      #...
    end
  end
end
```

Controllers

- should_assign_to
- should_not_assign_to
- should_set_the_flash_to
- should_not_set_the_flash
- should_set_the_flash_to
- should_redirect_to
- should_render_a_form
- should_render_template

RESTful Controllers

- Share a lot of common behavior
  - autorest, make_resourceful, resource_controller, resources_controller, ...
- Should be able to make assumptions for the basic actions: index, show, new, edit, create, update, destroy.
- Should be able to make those assumptions for HTML, XML, etc.

should_be_restful
...magic

```
class UsersControllerTest < Test::Unit::TestCase
  def setup
    @user = User.find(:first)
  end

  should_be_restful do |resource|
    resource.create.params = { :name => "Billy",
                               :party => 'Sure do!'}
    resource.update.params = { :name => "Changed" }
  end
end
```

should_be_restful
...magic

```
on GET to :show should assign @user.
on GET to :show should not set the flash.
on GET to :show should render 'show' template.
on GET to :show should respond with success.

on GET to :show as xml should assign @user.
on GET to :show as xml should have ContentType set to 'application/xml'.
on GET to :show as xml should respond with success.
on GET to :show as xml should return <user/> as the root element.

...about 50 more....
```

should_be_restful
...magic

Can be “configured” for most restful controllers

```
should_be_restful do |resource|
  resource.klass       = User
  resource.object      = :user
  resource.parent      = []
  resource.actions     = [:index, :show, :new, :edit,
                          :update, :create, :destroy]
  resource.formats     = [:html, :xml]

  resource.create.params = { :name => "bob",
                             :email => 'bob@bob.com'}
  resource.update.params = { :name => "sue" }

  resource.create.redirect  = "user_url(@user)"
  resource.update.redirect  = "user_url(@user)"
  resource.destroy.redirect = "users_url"

  resource.create.flash  = /created/i
  resource.update.flash  = /updated/i
  resource.destroy.flash = /removed/i
end
```

Too much magic?

- Making tests easier to write is good
- Generating short, simple tests is good
- Generating 50 tests with 5 lines...
  - Make sure you understand what's being tested
  - Be willing to write your own tests around should_be_restful

Shoulda internals

How Shoulda worked

How Shoulda works

How the macros are written

How to write your own

Contexts
naive implementation

should, context, setup, and teardown defined on Test::Unit (namespace pollution)

Bunch of class variables to keep track of contexts (ugly)

Broke when rails added a setup class method (generally buggy)

Contexts
much better

- Context class
  - should, setup, teardown, and context instance methods
- Test::Unit#should and #context
  - Just builds Context instances

Should statements
under the hood

- Creates a one-off context with a single should statement
- Inside a context bock:
  - Records the name and block
  - Test is built at the end of the context block.
  - Test runs setup/teardown blocks around should block

Macros
under the hood

```
class Test::Unit::TestCase
  # Ensures that the attribute cannot be set on update
  # Requires an existing record
  #
  #   should_protect_attributes :password, :admin_flag
  #
  def should_protect_attributes(*attributes)
    get_options!(attributes)
    klass = model_class

    attributes.each do |attribute|
      attribute = attribute.to_sym
      should "protect #{attribute} from mass updates" do
        protected = klass.protected_attributes
        assert protected.include?(attribute.to_s),
               "#{klass} is not protecting #{attribute}."
      end
    end
  end
end
```

Writing macros

Macros are totally simple with Shoulda

Just class methods that contain should statements or contexts

Writing macros

```ruby
# in test_helper.rb
def self.logged_in_as(user = :billy)
  context "When logged in as #{user}" do
    setup { @logged_in_user = login_as user }
    yield
  end
end
```

```ruby
# in your tests
logged_in_as(:admin) do
  should "be able to POST to #update" do
    #...
  end
end
```

General Testing Goodness

Mocking

Fixtures

Whitebox vs. Blackbox testing

Avoiding brittleness

Keep your tests effective

Mocking

Keeps your tests focused on the code at hand

Allows you to test integration with external resources

Can really improve the readability of a test

Speed

Overmocking

- Brittle tests
  - You can refactor a method correctly and still have failing tests.
- False sense of security
  - Be wary of integration points

```
User.expects(:find_by_sql).
     with("boogidy boo!").
     returns(@users)
```

Death to Fixtures!

Brittle: change a fixture and watch 200 tests fail

Bypass validations

Inexplicit: How many posts does users(:bob) have?

Encourages spaghetti

Generally unmaintainable

Alternatives to Fixtures

- Inline object creation
  - Nested contexts make this much more maintainable
- Extensive mocking (obvious dangers)
- http://b.logi.cx/2007/11/26/object-daddy
- Factories (my method of choice)

Factories

```ruby
module Factory
  def self.post_params(attrs = {})
    attrs[:body]   ||= "blah blah blah"
    attrs[:title]  ||= "Post title"
    attrs[:author] ||= (Author.find(:first) || Factory.create(:author))
    return attrs
  end

  # Shouldn't have to change stuff down here.
  def self.create(model, attrs = {})
    m = self.new(model, attrs)
    m.save!
    return m
  end

  def self.new(model, attrs = {})
    model.to_s.classify.constantize.new(send("#{model}_params", attrs))
  end
end
```

Factories

```
context "A admin with a bunch of posts" do
  setup do
    user = Factory.create(:user, :admin => true,
                                 :pet   => Factory.create(:pet))
    user.posts = [ Factory.create(:post),
                   Factory.create(:post) ]
  end

  context "when sent #approved_posts" do
    # ...
  end
end
```

White vs. Black
Whitebox Testing

Test the internals (mocking, testing private methods)

Shorter, more understandable tests

Easier to attain high test coverage

Can lead to overmocking

Brittle - refactoring your code will break your tests.

White vs. Black
Blackbox Testing

Test only the public API (call the method and test the results)

Ensures you're testing integration points

Won't break when you refactor

Tests are usually longer and harder to understand

Brittle tests

Brittle tests break due to trivial changes:

- Changing a method's implementation
- Changing unrelated parts of the application
- Changing other tests
- Running the tests in a different order

Brittle tests

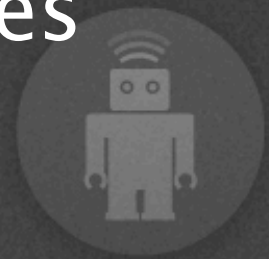
- Whitebox testing
  - Will break your tests when you refactor your internal implementation
  - May be worth it for test clarity

Brittle tests

- Overly explicit
  - Comparing entire contents of email
  - Too much assert_select – breaking views on copy changes
  - Comparing exact search results

Brittle tests

- Laziness
  - Assuming test order
  - Using data loaded from prior tests – accessing users(:billy) without loading the users fixture
  - Not cleaning up after creating files



Avoiding brittleness

- Make no assumptions
- Only describe the important behavior
- Keep tests short but fairly self-contained
  - Don't nest contexts too far
  - Describe one piece of behavior at a time
  - Don't use fixtures
- Beware of overmocking
- Favor blackbox testing

Effective tests

Be mindful of what you're testing.

Specify one piece of behavior at a time.

Names matter.

Describe expected behavior, not implementation details.

Avoid brittleness.

BDD

This is Behavior Driven Development

Future directions

Improve the ActiveRecord macros for edge cases

Add JSON and YAML support to should_be_restful

Maybe replace should_be_restful

Invite other maintainers

Git

More info

thoughtbot.com/projects/shoulda

rdocs: dev.thoughtbot.com/shoulda

Send any questions or suggestions to the google group: shoulda@googlegroups.com or groups.google.com/group/shoulda

Submit patches/bugs to tammer.lighthouseapp.com/projects/5807

Thanks

http://thoughtbot.com

(we're hiring)

?