How to test my app

Rspec has a nice syntax. However, it's not rails default. But, listen :

I discover that **minitest** has a **Minitest::Spec** module providing a **rspec-like syntax**.

See **Annexe** to learn using tests when devise authentication is on the flow...

Natively

You only need to require the module.

```
# any test file
require 'test_helper'
require 'minitest/spec'
```

That's it !

Choosing between test syntax or spec syntax is a matter of taste. The former tests the validity of assertions; the later require specifications to have expected results.

So :

```
# instead of writing
class MyClassTest < Minitest::TestCase
  test 'provide a Capitalized string' do
    assert_equal 'Hello World', cap_it( 'hello world' )
  end
end

# you write
describe MyClass do
  it 'successfully capitalize a given string' do
    _( cap_it( 'hello world' ) ).must_equal 'Hello World'
  end
end
```

I think the main interest is in `describe` line. No need to explicitely define a class subclassing the right superclass.

A gem simplify the use of Minitest::Specs : `minitest-spec-rails` . No need to require 'minitest/spec'...

Expectations

Remarque : take care to call `_(obj).must_xxx` rather than `obj.must_xxx` to avoid issues with threaded tests. Equivalent syntaxes are `expect(obj).must_xxx` and `value(obj).must_xxx` but `_` gives the more readable one (because `expect something must be` do not sound like good english!)

`_` can also receive a block : `_ { .. }.must_xxx` (as `expect` or `value`).

Someone could provide new expectation syntax by adding them in the `Minitest::Expectations` module to fill my needs :

```
module Minitest::Expectations # Monkey patching, put it inside test_helper.rb
  infect_an_assertion :assert, :must_be_true: reverse
end
```

Defaults

```
value(obj).must_be(operator, expected) # for example, 10.must_be :<, 11
value(obj).must_be_close_to # the equivalent of assert_in_delta
value(obj).must_be_empty # Fails unless obj.empty?
value(obj).must_be_instance_of(klass) # Fails unless obj.class == klass
value(obj).must_be_kind_of(klass or Module) # Fails unless obj is of class klass or klass is one of it
value(obj).must_be_nil
value(obj).must_be_same_as # tests for true object equality
lambda {}.must_be_silent
value(obj).must_be_within_delta
value(obj).must_be_within_epsilon
value(obj).must_equal(other) # Does a ==/eql? comparison between two objects.
value(obj).must_include(other)
value(obj).must_match(regex) # A regular expression match, e.g.
"hello".must_match /w+/
lambda {}.must_output(stdout, [stderr..]) # The block should have certain
output on stdout or stderr. Set stdout to nil just to check stderr.
lambda {}.must_raise(exception)
value(obj).must_respond_to(method)
value(obj).must_throw(sym)

# The above are all positive valueations but the opposite ones are easy to build
# as in most cases you can switch must with wont. For example:
wont_be, wont_be_empty, wont_be_instance_of, wont_be_kind_of
wont_be_nil, wont_be_same_as, wont_equal,
wont_include, wont_match, wont_respond_to
```

What and how to test ?

Main guide : a lot of `isolated, unit-tests` . Also called functional tests. They test object (or method) return regarding to given arguments. All dependencies are faked. A test have to suppose other things work and do not have to test they really are. So use `stubs` and `mocks` .

Here we see how dependency injection easy the testing part :

```
class Car
  attr_reader :engine

  def initialize( engine )
    @engine = engine
  end

  # method to test
  def status
    if engine.start?
      "Engine started"
    else
      "Engine stopped"
    end
  end
end
```

We want to test `my_car` method but not the `start?` method on engine. So we introduce a faked engine `start?` return.

```
# with stubbing
describe Car do
  it 'correctly reacts to engine start' do
    @engine = Engine.new
    @car = Car.new( @engine )
    @engine.stub :start?, true do
      _( @car.status ).must_equal "Engine started"
    end
    @engine.stub :start?, false do
      _( @car.status ).must_equal "Engine stopped"
    end
  end
end
```

```
# with mocking we can test if `start?` was called, with eventually the right
arguments
describe Car do
  it 'correctly reacts to engine start' do
    @engine = Minitest::Mock.new
    @car = Car.new( @engine )

    @engine.expect :start?, true
    _( @car.status ).must_equal "Engine started"
    @engine.verify

    @engine.expect :start?, false
      _( @car.status ).must_equal "Engine stopped"
    @engine.verify
  end
end
```

Controller testings

They're testable in two ways. First are functional tests, second are integration tests.

Controller functional tests

It is not the place to test db access nor the view. I only have to test behaviour, given `params` , `cookies` and `session` hashes (together with `current_user` devise method). And the behaviour is either `:success` , `:redirect` and eventually the `path` .

So mocking (or only stubbing) all db-relative stuff is essential :

- For `C_UD` operations, stub the processor :

```
InputsProcessor.stub :call, true do# same with false
  get :create# or :update or :delete
  _( response.status ).must_equal :success # same with :redirect
end
```

Here a way to go :

```
# budget_controller.rb
def update
  # even if update returns updated budget instance, this side effect is explicit..
  unless budget = BudgetsProcessor.(
      update: current_budget, context: { params: params }
  )
    redirect_to :edit_budget
  else
    set_session_hash_for budget
    redirect_to :authenticated_home
  end
end

# budgets_processor.rb
def update
  target.update( permits_params ) ? target : false
end
```

```
# budget_controller_test.rb
describe BudgetController do
  include Devise::Test::ControllerHelpers

  describe '#update' do
    before do
      @user = users(:user_01)
      sign_in @user
      @budget = budgets( :budget_1 )
      @new_budget = Budget.new(
        @budget.attributes.merge(
          currency_unit: "new_#{@budget.currency_unit}"
        )
      )
      @params = { id: @budget.id, budget: @new_budget.attributes }
      @session = {}.merge( session ) # copy needed
    end

    it "calls BudgetsProcessor.( :update, ... ) with adequat arguments" do
      processor = Minitest::Mock.new
      processor.expect :call, @new_budget do |**args|
        _( args.keys.first ).must_equal :update, "Wrong call to processor##{args.keys.first}"
        _( args[args.keys.first] ).must_equal @budget, "Wrong target"
        _( args[:context].keys ).must_include :params, "params not given"
      end
      BudgetsProcessor.stub :call, processor do
```

```
        patch :update, params: @params
      end
      processor.verify
    end

    describe "on successful update" do
      it "updates session hash and redirect to :authenticated_home" do
        BudgetsProcessor.stub :call, @new_budget do
          patch :update, params: @params
          _( session[:currency_unit] ).must_equal @new_budget.currency_unit
          _( patch :update, params: @params ).must_redirect_to :authenticated_home
        end
      end
    end

    describe "on unsuccessful update" do
      it "does not update session hash and redirect to :edit" do
        BudgetsProcessor.stub :call, false do
          patch :update, params: @params
          _( session[:currency_unit] ).must_equal @budget.currency_unit
          _( patch :update, params: @params ).must_redirect_to :edit_budget
        end
      end
    end
  end
end
```

- For `_R__` operation, mock the collector :

```
def home # is there anything to test here ?
  @entries = HomePresenter.( EntriesCollector.( :query_for_home, context: {} ) )
end
```

```
# simple stubbing
fake_entries = []
fake_entries << Entry.new( ... ) # needed for the view to not raise errors..
EntriesCollector.stub :call, fake_entries do
  _( get :home ).must_equal 200
end

# mocking allow argument tests : Hey maybe over controller test area !!
fake_collector = Minitest::Mock.new
fake_return = []
Bay.each { |bay| fake_return << Entry.new( bay_id: bay.id ) }

fake_collector.expect :call, fake_return do |query_name, **context|
  _( query_name ).must_equal :query_for_home, "Wrong collector query name"
  _( context ).must_be_instance_of Hash, "Context is #{context}; not a hash"
  unless context.empty?# demo purpose because in a given method context presence is known
    _( context.keys ).must_include :context, "Do not provide a context"
    _( context[:context].keys ).must_include :budget_id
  end
  true # needed because conditional returns !context.empty?
end

EntriesCollector.stub :call, fake_collector do
  _( get :home ).must_equal 200
end
collector.verify
```

`fake_entries` have to respond to view API. So it is (also) a view test; this is not desirable. Hence the question : do I have to test these methods ?

Controller integration tests

It may be needed to follow annexe configuration. This paragraph is to develop when I'll have experienced this kind of tests.

Annexe

Configure tests so devise authentication lets you operate tests..

- First, complete the monkey patching of `class ActiveSupport::TestCase` :

```
# test/test_helper.rb
# ------------------------------------------------
# for devise
include Devise::Test::IntegrationHelpers
include Warden::Test::Helpers

def log_in( account )
  if integration_test?
    # use warden helper
    login_as( account, :scope => :account )
  else # controller_test, model_test
    # use devise helper
   sign_in( account )
  end
end
# ------------------------------------------------
```

- Second, complete your controller tests (or whatever need an authenticated user)

```
# stays_controller_test.rb
class StaysControllerTest < ActionDispatch::IntegrationTest
  include Devise::Test::IntegrationHelpers

  #------------------------------------------------
  # for devise user
  setup do
    get '/accounts/sign_in'
    sign_in account( :account_01 )
    post account_session_url

    # if you want to test that things are working correctly, uncomment below
    # follow_redirect!
    # assert_response :success
  end
  #------------------------------------------------

  test "the truth" do
    assert true
  end
end
```

- Third, create an account fixture

```
# test/fixtures/accounts.yml
account_01:
  id: 1
  email: mazu@sfr.fr
  encrypted_password: <%= Devise::Encryptor.digest( User, 'tttttttt') %>
```