Testing using MiniTest in Ruby

Overview

In this tutorial, we will create some tests in Ruby using Minitest and the "minitest-reporters" gem.

The "minitest-junit" could also be used, with minor changes to the code below, particularly in the "test_helper.rb".

Requirements

- Install minitest and minitest-reporters gem (or the "minitest-junit" gem as an alternative)

```
gem install minitest minitest-reporters
```

Description

The code that follows is from the minitest github project.

Let's start by using a sample Ruby class.

lib/meme.rb

```
class Meme
  def i_can_has_cheezburger?
    "OHAI!"
  end
  def will_it_blend?
    "YES!"
  end
end
```

The tests will require some common code that can be written in a "test_helper" Ruby script.

test/test_helper.rb

```
$LOAD_PATH.unshift File.expand_path('../lib', __FILE__)
require 'meme'
require 'minitest/autorun'
require "minitest/reporters"
Minitest::Reporters.use! Minitest::Reporters::JUnitReporter.new
```

Writing some unit tests is straightforward.

test/meme_unit_test.rb

```
require 'test_helper'
class MemeTest < Minitest::Test
  def setup
    @meme = Meme.new
  end
  def test_that_kitty_can_eat
    assert_equal "OHAI!", @meme.i_can_has_cheezburger?
  end
  def test_that_it_will_not_blend
    refute_match /^no/i, @meme.will_it_blend?
  end
  def test_that_will_be_skipped
    skip "test this later"
  end
end
```

MiniTest also supports RSpec-like features such as the ability to use "specs". In this case, the test is wrapped inside one or multiple "describe" blocks, and is consubstantiated in an "it" block.

test/meme_spec_test.rb

```
require 'test_helper'
describe Meme do
  before do
    @meme = Meme.new
  end
  describe "when asked about cheeseburgers" do
    it "must respond positively" do
      @meme.i_can_has_cheezburger?.must_equal "OHAI!"
    end
  end
  describe "when asked about blending possibilities" do
    it "won't say no" do
      @meme.will_it_blend?.wont_match /^no/i
    end
  end
end
```

The two different approaches are valid and their results should be similar.

After running the tests and generating the JUnit XML report(s), it/they can be imported to Xray (either by the REST API or through the **Import Execution Results** action within the Test Execution).

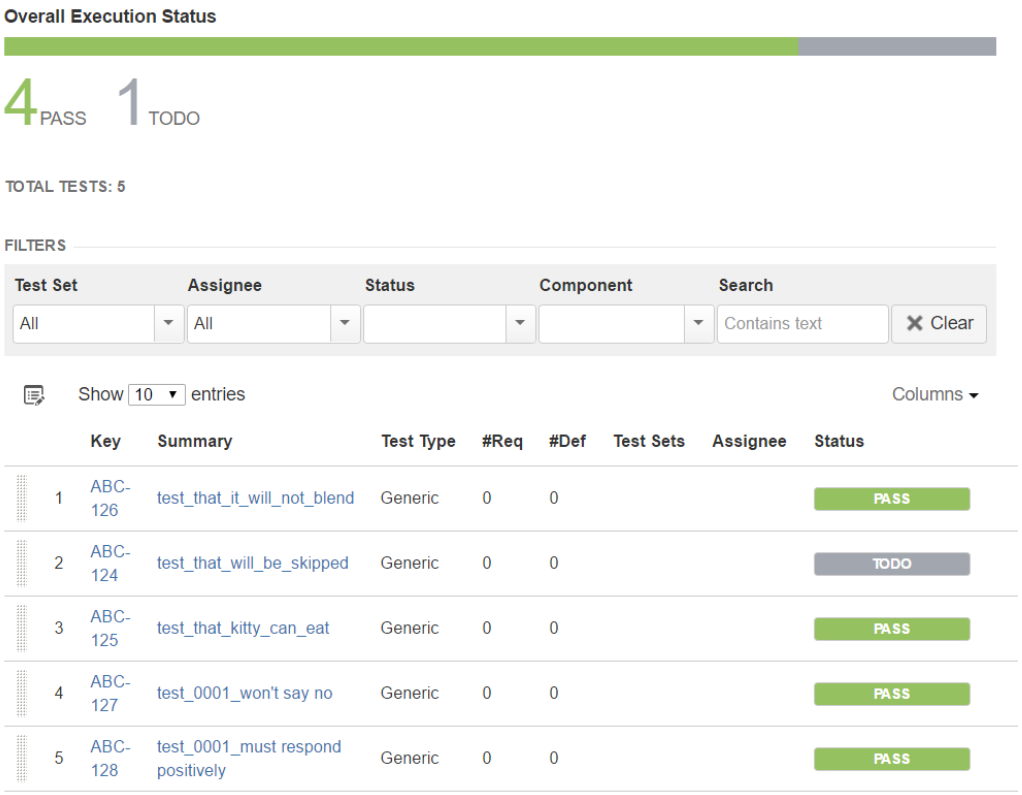
```
rake test
```

Several JUnit XML will be produced:

- TEST-MemeTest.xml
- TEST-Meme-when-asked-about-blending-possibilities.xml
- TEST-Meme-when-asked-about-cheeseburgers.xml

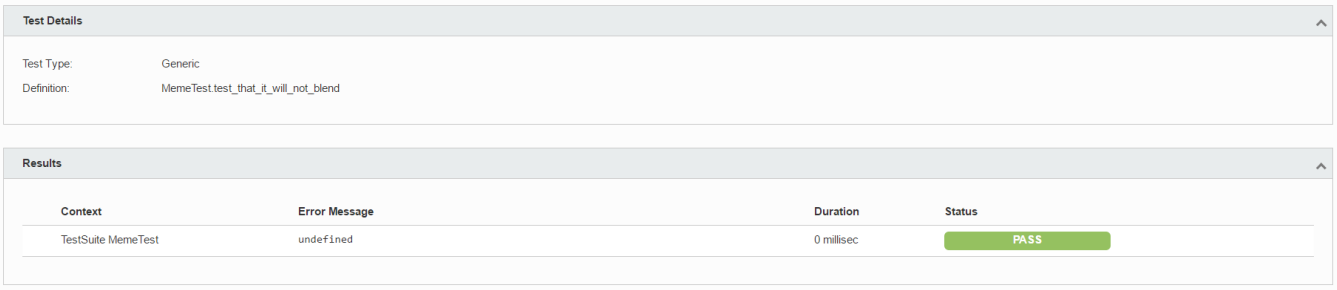
TEST-MemeTest.xml contains the results of the unit tests. The other two files contain results related to the "spec" test.

Note: This example could be further optimized to obtain just a JUnit XML file containing the results for the two different test classes.

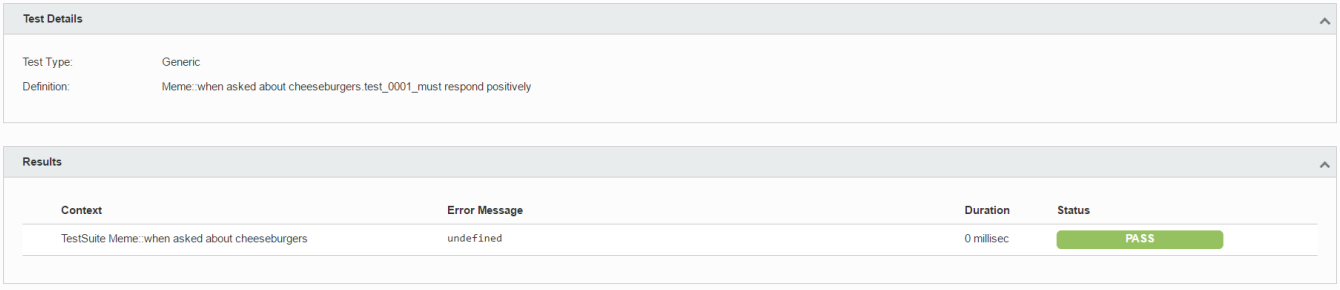


The test is mapped to a Generic Test in Jira, and the **Generic Test Definition** field contains the name of Ruby test class concatenated with the name of the method that implements the test case.

The Execution Details of the Generic Test contains information about the Test Suite, which in the case of the unit tests, corresponds to the name of the class.



For the spec-related tests, they're mapped in a slightly different way: the multiple "describe" are concatenated using "::", along with the name of the "it" block, preceded by "test_<counter>".



References

- https://github.com/seattlerb/minitest
- https://github.com/kern/minitest-reporters
- https://github.com/aespinosa/minitest-junit
- https://github.com/Devskiller/devskiller-sample-ruby-calculator