TEST DRIVING RAILS

TAKING MINITEST AND FIXTURES FOR A SPIN

1st edition

By Josef Strzibny

Test Driving Rails

Taking Minitest and Fixtures for a spin

Josef Strzibny

Version Preview 1

Table of Contents

Rails blends Minitest and fixtures together in a seemless integration that gives developers testing super powers out-of-the-box. This guidebook is my opinionated take on testing Ruby on Rails applications with these defaults.

NOTE	Rails version used in this revision is `7.2`.

@ Josef Strzibny

Chapter 1. Introduction

Have you ever worked on completely untested applications? It can be incredibly painful. Without automated tests, it's hard to ensure stability with code changes, protect production data, minimize the impact of bugs, and reduce the time needed for manual testing.

Automated testing gives us a fast development and deployment cycle. It gives us the confidence to ship more in a shorter amount of time. Luckily, Rails provides us with seamless Minitest integration and a way to create test data with fixtures.

Despite Minitest prominence in Ruby, Rails, and eco-system libraries, the industry overwhelmingly adopted RSpec for testing. And I felt the art for default Rails tests was lost.

I have stayed true and loyal to Minitest for all of my projects over the years. Minitest is an important item on Rails Doctrine omasake menu and I want to help bring back the spark for it with this book.

Minitest test cases are straightforward to write and more importantly effortless to read later. They are just a piece of Ruby code.

So let's go and take Rails for a test drive.

Chapter 2. Minitest

Created by Ryan Davis in 2008 as a faster and more flexible alternative to Ruby's own Test::Unit. Minitest replaced Test::Unit later in 2011 in Ruby 1.9 and it's Ruby's and Rails' default test framework. Over time more features were added, including a rich set of assertions, mocks and stubs, benchmark capabilities, or support for parallel testing.

Before we have a look at all of that, let's start by looking at how to write and run basic Minitest tests.

2.1. Tests and Specs

Minitest is mostly known for simple test cases subclassed from `Minitest::Test` which provides a lightweight unit testing framework. Here's an example of a basic Minitest test:

```
require "minitest"

class LobsterRoll
  def size
    :big
  end

  def delicious?
    true
  end
end

class LobsterRollTest < Minitest::Test
  def test_is_delicious
    roll = LobsterRoll.new
    assert roll.delicious?
  end
end
```

Any method of `Minitest::Test` subclass started with `test_` is turned into runnable test cases automatically. The only other thing needed is at least one simple assertion to produce a test report.

A single test is usually composed of three steps. We prepare the test environment in a setup step before running the test case and reset the environment as part of a final teardown. The setup step prepares any data or environment needed in tests or specs. In the case of a Rails application, this also involves steps done by the Rails framework, namely loading your fixtures.

Here's an example of a `setup` step preparing an object for the test that follows:

```
class LobsterRollTest < Minitest::Test
```

```
  def setup
    @roll = LobsterRoll.new
  end
  ...
```

If you are coming from RSpec, `subject` and `let!` blocks would also be part of this simple method. There is no conditional object creation so you should include things relevant for all tests in the same class.

A test case calls the object methods under tests and the result is verified with assertions:

```
class LobsterRollTest < Minitest::Test
  ...
  def test_name
    assert @roll.delicious?
    assert_equal :size, @roll.size
  end
  ...
```

The simplest of assertions are `assert` and `assert_equal`. One asserts truthiness and the other equality. There are many more assertions which we'll see in the Assertions chapter, but we can test most things with these two.

Finally, there is a teardown step that should clean up the environment for the next test. This means clearing the database of the records we created or verifying mocks. If the lobster roll is an object in the database, we might want to remove it at the end:

```
require "minitest"

class LobsterRoll < ActiveRecord::Base
end

class LobsterRollTest < Minitest::Test
  def setup
    @roll = LobsterRoll.create!
  end
  ...
  def teardown
    @roll.destroy!
  end
  ...
```

We'll see that this is usually not needed since Rails and other Minitest libraries implement a teardown on their own. They can take advantage of Minitest lifecycle hooks to extend the tests:

```ruby
module MyMinitestPlugin
  def before_setup
    super
  end

  def after_setup
    super
  end

  def before_teardown
    super
  end

  def after_teardown
    super
  end
end

class MiniTest::Test
  include MyMinitestPlugin
end
```

The above tests can also be represented with `Minitest::Spec`, a layer of syntactic sugar allowing for RSpec-like specification way of testing:

```ruby
require "minitest/spec"

describe "LobsterRoll" do
  it "is delicious" do
    roll = LobsterRoll.new
    expect(roll.delicious?).must_equal true
  end
end
```

Similarly, Rails provides its syntactic sugar for `Minitest::Test` as we'll see later. In this book, I'll stick to the Rails style of tests, but if you like specs more, it's an option.

2.2. Runner

Once the test is written it's time to run it. Minitest suite is run with a call to `Minitest.run` which accepts list of command line arguments:

```ruby
!#/usr/bin/env ruby
```

```ruby
require 'minitest'

class ExampleTest < Minitest::Test
  def test_truth
    assert true
  end
end

Minitest.run(ARGV)
```

Running this test prints the following:

```
% ruby example.rb
Run options: --seed 56077

# Running:

.

Finished in 0.000289s, 3460.2454 runs/s, 3460.2454 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

We can see one passed run with 1 assertion. And we can also see an imlicit `--seed` option. Since tests are randomized by default, providing the seed makes it possible to rerun the test suite in the same order.

This brings us to `ARGV` which refers to arguments passed to the Ruby script:

```
$ ruby test.rb -n test_truth
Run options: -n test_truth --seed 33669

# Running:

.

Finished in 0.000230s, 4347.8018 runs/s, 4347.8018 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

Now only the `test_trush` test case will be run.

We can also adjust the arguments from within the script:

```ruby
!#/usr/bin/env ruby

require "minitest"

class ExampleTest < Minitest::Test
  def test_truth
    assert true
  end

  def test_falsehood
    refute false
  end
end

ARGV << "--verbose"

# Will only run test_truth
Minitest.run(ARGV)

# Or with fixed arguments
Minitest.run(["-n", "test_truth", "--verbose"])
```

Minitest also features a handy `autorun` method that will do the above for you using the `Kernel.at_exit` hook:

```ruby
# test.rb
require "minitest"
...
Minitest.autorun
```

Since it's registered at the exit it's more common to see it used as `require` call at the top:

```ruby
# test.rb
require "minitest/autorun"
...
class ExampleTest < Minitest::Test
  def test_truth
    assert true
  end
end
```

And here's the source for `Minitest.autorun` method itself:

```
# from Minitest source
def self.autorun
  if Object.const_defined?(:Warning) && Warning.respond_to?(:[]=)
    Warning[:deprecated] = true
  end

  at_exit {
    next if $! and not ($!.kind_of? SystemExit and $!.success?)

    exit_code = nil

    pid = Process.pid
    at_exit {
      next if !Minitest.allow_fork && Process.pid != pid
      @@after_run.reverse_each(&:call)
      exit exit_code || false
    }

    exit_code = Minitest.run ARGV
  } unless @@installed_at_exit
  @@installed_at_exit = true
end
```

We can see that this registers Ruby's `at_exit` hook and runs `Minitest.run` in the end. But what about the `Minitest.allow_fork` check? Minitest supports forking in tests so it has to skip the `at_exit` registration in such a case.

Here's one example of a test like that:

```
require "minitest/autorun"

class ForkingTest < Minitest::Test
  def setup
    Minitest.allow_fork = true
  end

  def test_forking
    pid = fork do
      assert true
    end
    Process.wait(pid)
  end
end
```

Minitest also features built-in parallelization. To enable it we have to call `parallelize_me!` in each test

class:

```
require "minitest/autorun"

class ExampleTest < Minitest::Test
  parallelize_me!

  def test_example_one
    puts Process.pid
    puts Thread.current.object_id
    assert_equal 2, 1 + 1
  end

  def test_example_two
    puts Process.pid
    puts Thread.current.object_id
    assert_equal 4, 2 + 2
  end
end
```

NOTE	To set parallelization for the whole test suite at once, we can call `Minitest::Test.parallelize_me!`.

Printing `Process.pid` and `Thread.current.object_id` reveal that this parallelization is done by default using threads rather than processes. This is generally a good setting if the tests are not CPU-bound or run on a platform like JRuby.

We can control how many threads are used by setting the `MT_CPU` environment variable:

```
$ MT_CPU=4 ruby parallel_test_suite.rb
```

If we would like to parallelize the test suite using processes with plain Minitest, we have to go for a plugin such as `minitest-parallel_fork`. One good reason for that is to avoid issues with modified constants.

Rails test stack builds on top of Minitest and already comes with its options for parallelization, including process-based parallelization. We'll get back to it a bit later.

2.3. Reporters

To get the test results out Minitest.run, Minitest utilizes various reporters that record single test results that can be merged into a final report.

As an example, ProgressReporter implements the dots you'll see on the console while running a

Minitest test suite, StatisticsReporter handles timing and counters, and SummaryReporter provides the summary at the end of a test run.

We can instruct Minitest to use any different reporter with `Minitest::Reporters.use!`:

```
gem "minitest-reporters"

# test_helper.rb or spec_helper.rb
require "minitest/reporters"
Minitest::Reporters.use!(Minitest::Reporters::SpecReporter.new)
```

Suddenly the final report layout completely changes:

```
$ ruby test.rb
Started with run options -n test_truth --seed 31890

ExampleTest
  test_truth                                                    PASS (0.00s)

Finished in 0.00030s
1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

The abstract reporter interface features `start`, `record`, `report`, and `passed?` methods in case you would like to implement a custom reporter.

We don't need to configure any special reporting, but it's simply one of the many options for how to extend Minitest with plugins.

Chapter 3. Fin

You reached the end of the preview.

Thanks for your interestest in Test Driving Rails. If you would like to purchase the whole book, you can do it at https://testdrivingrails.com.