Extracted from:

Effective Testing with RSpec 3

Build Ruby Apps with Confidence

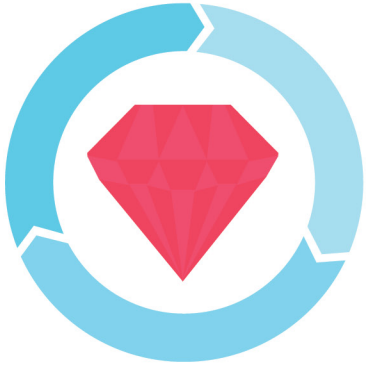
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Effective Testing with RSpec 3

Build Ruby Apps with Confidence



Myron Marston
and Ian Dees
edited by Jacquelyn Carter

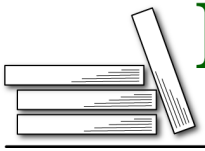
Foreword by Tom Stuart,
author of *Understanding Computation*

Effective Testing with RSpec 3

Build Ruby Apps with Confidence

Myron Marston
Ian Dees

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-198-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2017

Testing the Invalid Case

So far, our integration spec checks only the "happy path" of saving a valid expense. Let's add a second spec that tries an expense missing a payee:

06-integration-specs/07/expense_tracker/spec/integration/app/ledger_spec.rb

```
context 'when the expense lacks a payee' do
  it 'rejects the expense as invalid' do
    expense.delete('payee')

    result = ledger.record(expense)

    expect(result).not_to be_success
    expect(result.expense_id).to eq(nil)
    expect(result.error_message).to include('`payee` is required')

    expect(DB[:expenses].count).to eq(0)
  end
end
```

Here, the Ledger instance should return the correct failure status and message, and the database should have no expenses in it.

Since this behavior isn't implemented, we want this new spec to fail:

```
$ bundle exec rspec spec/integration/app/ledger_spec.rb
« truncated »

Failures:

  1) ExpenseTracker::Ledger#record when the expense lacks a payee rejects
  the expense as invalid

« truncated »

  2) ExpenseTracker::Ledger#record with a valid expense successfully saves
  the expense in the DB

« truncated »

Finished in 0.02597 seconds (files took 0.18213 seconds to load)
2 examples, 2 failures

Failed examples:

rspec ./spec/integration/app/ledger_spec.rb:34 #
ExpenseTracker::Ledger#record when the expense lacks a payee rejects the
expense as invalid
```

```
rspec ./spec/integration/app/ledger_spec.rb:20 #                    ↩
ExpenseTracker::Ledger#record with a valid expense successfully saves the   ↩
expense in the DB

Randomized with seed 57045
```

That's strange. *Both* specs failed. Did we break any other specs outside the two we're running here? Try rerunning the entire suite a few times. Your exact output will differ from ours—you might see two failures or just one. Here's a test run with two failures:

```
$ bundle exec rspec
« truncated »

Finished in 0.06926 seconds (files took 0.21812 seconds to load)
11 examples, 2 failures, 1 pending

Failed examples:

rspec ./spec/integration/app/ledger_spec.rb:20 #                    ↩
ExpenseTracker::Ledger#record with a valid expense successfully saves the   ↩
expense in the DB
rspec ./spec/integration/app/ledger_spec.rb:34 #                    ↩
ExpenseTracker::Ledger#record when the expense lacks a payee rejects the    ↩
expense as invalid

Randomized with seed 32043
```

The output is slightly different each time you run it, because RSpec is running the specs in random order. This technique—enabled via the `config.order = :random` line in `spec_helper.rb`—is useful for finding *order dependencies*; that is, specs whose behavior depends on which one runs first.

Test in Random Order to Find Order Dependencies

If your specs run in the same order every time, you may have one that's only passing because an earlier, broken one left some state behind. Use your test suite's random ordering option to surface these dependencies.

A *random seed* gives you a specific, repeatable test order. You can replay any such sequence by passing the `--seed` option to RSpec along with the seed number reported in the output.

For example, RSpec's output tells us that the previous run was using seed 32043. You can replay that specific test order any time you want:

```
$ bundle exec rspec --seed 32043
« truncated »

Finished in 0.05941 seconds (files took 0.22746 seconds to load)
11 examples, 2 failures, 1 pending
```

```
Failed examples:

rspec ./spec/integration/app/ledger_spec.rb:20 # ExpenseTracker::Ledger#record with a valid expense successfully saves the expense in the DB
rspec ./spec/integration/app/ledger_spec.rb:34 # ExpenseTracker::Ledger#record when the expense lacks a payee rejects the expense as invalid

Randomized with seed 32043
```

While RSpec isn't able to tell you *why* the ordering dependency is happening, it can certainly help you identify *which* specs you need to run to reproduce it.

With the --bisect option, RSpec will systematically run different portions of your suite until it finds the smallest set that triggers a failure:

```
$ bundle exec rspec --bisect --seed 32043
Bisect started using options: "--seed 32043"
Running suite to find failures... (0.45293 seconds)
Starting bisect with 2 failing examples and 9 non-failing examples.
Checking that failure(s) are order-dependent... failure appears to be order-dependent

Round 1: bisecting over non-failing examples 1-9 . ignoring examples 1-5 (0.45132 seconds)
Round 2: bisecting over non-failing examples 6-9 . ignoring examples 6-7 (0.43739 seconds)
Round 3: bisecting over non-failing examples 8-9 . ignoring example 8 (0.43102 seconds)
Bisect complete! Reduced necessary non-failing examples from 9 to 1 in 1.64 seconds.

The minimal reproduction command is:
  rspec './spec/acceptance/expense_tracker_api_spec.rb[1:1]'
  './spec/integration/app/ledger_spec.rb[1:1:1:1,1:1:2:1]' --seed 32043
```

RSpec has given us a minimal set of specs we can run any time to see the failure:

```
$ bundle exec rspec './spec/acceptance/expense_tracker_api_spec.rb[1:1]' './spec/integration/app/ledger_spec.rb[1:1:1:1,1:1:2:1]' --seed 32043
Run options: include {:ids=>{"./spec/acceptance/expense_tracker_api_spec.rb"=>["1:1"], "./spec/integration/app/ledger_spec.rb"=>["1:1:1:1", "1:1:2:1"]}}

Randomized with seed 32043
*FF

« truncated »

Finished in 0.0485 seconds (files took 0.21859 seconds to load)
3 examples, 2 failures, 1 pending
```

```
Failed examples:

rspec ./spec/integration/app/ledger_spec.rb:20 # ExpenseTracker::Ledger#record with a valid expense successfully saves the expense in the DB
rspec ./spec/integration/app/ledger_spec.rb:34 # ExpenseTracker::Ledger#record when the expense lacks a payee rejects the expense as invalid

Randomized with seed 32043
```

The numbers in square brackets are called *example IDs*; they indicate each example's position in its file, relative to other examples and nested groups. For example, some_spec.rb[2:3] would mean "the third example inside the second group in some_spec.rb." You can paste these values into your terminal just as you did with line numbers in *Running Specific Failures*, on page ?.

With this set of specs, we can see that the new spec causes the prior spec to fail if the new one runs first. Our database writes are leaking between tests.

Watch for Test Interaction in Integration Specs

Your integration specs interact with external resources: the file system, the database, the network, and so on. Because these resources are shared, you have to take extra care to restore the system to a clean slate after each spec.

Isolating Your Specs Using Database Transactions

To solve this issue, we're going to wrap each spec in a database transaction. After each example runs, we want RSpec to *roll back* the transaction, canceling any writes that happened and leaving the database in a clean state. Sequel provides a test-friendly method for wrapping code in transactions.[5]

An RSpec around hook would be perfect for this task. You already have a spec/support/db.rb file for database support code, so add it there, inside the RSpec.configure block:

06-integration-specs/07/expense_tracker/spec/support/db.rb

```
c.around(:example, :db) do |example|
  DB.transaction(rollback: :always) { example.run }
end
```

The sequence of calls in the new hook is a bit twisty, so bear with us for a second. For each example marked as requiring the database (via the :db tag; you'll see how to use this in a moment), the following events happen:

5. http://sequel.jeremyevans.net/rdoc/files/doc/testing_rdoc.html

1. RSpec calls our around hook, passing it the example we're running.
2. Inside the hook, we tell Sequel to start a new database transaction.
3. Sequel calls the inner block, in which we tell RSpec to run the example.
4. The body of the example finishes running.
5. Sequel rolls back the transaction, wiping out any changes we made to the database.
6. The around hook finishes, and RSpec moves on to the next example.

We only want to spend time setting up the database when we actually use it. Although starting and rolling back a transaction takes less than a tenth of a second, that dwarfs the runtime of our fast, focused unit specs.

This is where the notion of *tagging* comes in. A tag is a piece of metadata—custom information—attached to an example or group. Here, you'll use a new symbol, :db, to indicate that an example touches the database:

06-integration-specs/07/expense_tracker/spec/integration/an_integration_spec.rb

```
require_relative '../support/db'

RSpec.describe 'An integration spec', :db do
  # ...
end
```

Notice that we've had to do two things to make sure this spec runs inside a database transaction:

- Explicitly load the setup code from support/db
- Tag the example group with :db

If you have several spec files that touch the database, it's easy to forget to do one or the other of these things. The result can be specs that pass or fail inconsistently (depending on when and how you run them).

It'd be nice to be able to tag the database-related example groups with :db and trust that the support code will be loaded as needed. Luckily, RSpec has an option that supports exactly this usage. In your spec_helper.rb, add the following snippet inside the RSpec.configure block:

06-integration-specs/07/expense_tracker/spec/spec_helper.rb

```
RSpec.configure do |config|
  config.when_first_matching_example_defined(:db) do
    require_relative 'support/db'
  end
```

With that hook in place, RSpec will *conditionally* load spec/support/db.rb if (and only if) any examples are loaded that have a :db tag.

Now, you'll need to go add the tag to your specs. Both your acceptance and integration specs need it at the end of the RSpec.describe line, just before the do keyword. First, spec/acceptance/expense_tracker_api_spec.rb:

06-integration-specs/08/expense_tracker/spec/acceptance/expense_tracker_api_spec.rb
```
RSpec.describe 'Expense Tracker API', :db do
```

Next, spec/integration/app/ledger_spec.rb:

06-integration-specs/08/expense_tracker/spec/integration/app/ledger_spec.rb
```
RSpec.describe Ledger, :aggregate_failures, :db do
```

While you're at it, you can also remove the require_relative '../../support/db' line from ledger_spec.rb.

At last, you can run RSpec with the same random seed and get just the single failure we were expecting:

```
$ bundle exec rspec --seed 32043
« truncated »

Finished in 0.05786 seconds (files took 0.22818 seconds to load)
11 examples, 1 failure, 1 pending

Failed examples:

rspec ./spec/integration/app/ledger_spec.rb:33 #                    ↩
ExpenseTracker::Ledger#record when the expense lacks a payee rejects the    ↩
expense as invalid

Randomized with seed 32043
```

We're back down to just one spec failing predictably. It's time to continue the Red/Green/Refactor cycle by adding just enough behavior to your object to make the spec pass.