The RSpec Style Guide

Table of Contents

Introduction . . . 1
How to Read This Guide . . . 2
A Living Document . . . 2
Layout . . . 2
Example Group Structure . . . 8
Example Structure . . . 14
Naming . . . 25
Matchers . . . 29
Rails: Integration . . . 31
Rails: Views . . . 32
Rails: Controllers . . . 34
Rails: Models . . . 36
Rails: Mailers . . . 39
Recommendations . . . 40
Related Guides . . . 40
Contributing . . . 40
License . . . 41
Credit . . . 41

Introduction

> Role models are important.
>
> — Officer Alex J. Murphy / RoboCop

This RSpec style guide outlines the recommended best practices for real-world programmers to write code that can be maintained by other real-world programmers.

RuboCop, a static code analyzer (linter) and formatter, has a `rubocop-rspec` extension, provides a way to enforce the rules outlined in this guide.

NOTE This guide assumes you are using RSpec 3 or later.

You can generate a PDF copy of this guide using AsciiDoctor PDF, and an HTML copy with AsciiDoctor using the following commands:

```
# Generates README.pdf
asciidoctor-pdf -a allow-uri-read README.adoc

# Generates README.html
asciidoctor
```

TIP

Install the `rouge` gem to get nice syntax highlighting in the generated document.

```
gem install rouge
```

How to Read This Guide

The guide is separated into sections based on the different pieces of an entire spec file. There was an attempt to omit all obvious information, if anything is unclear, feel free to open an issue asking for further clarity.

A Living Document

Per the comment above, this guide is a work in progress - some rules are simply lacking thorough examples, but some things in the RSpec world change week by week or month by month. With that said, as the standard changes this guide is meant to be able to change with it.

Layout

Empty Lines inside Example Group

Do not leave empty lines after `feature`, `context` or `describe` descriptions. It doesn't make the code more readable and lowers the value of logical chunks.

```ruby
# bad
describe Article do

  describe '#summary' do

    context 'when there is a summary' do

      it 'returns the summary' do
        # ...
      end
    end
  end
end

# good
describe Article do
  describe '#summary' do
    context 'when there is a summary' do
      it 'returns the summary' do
        # ...
      end
    end
  end
end
```

Empty Line between Example Groups

Leave one empty line between `feature`, `context` or `describe` blocks. Do not leave empty line after the last such block in a group.

```ruby
# bad
describe Article do
  describe '#summary' do
    context 'when there is a summary' do
      # ...
    end
    context 'when there is no summary' do
      # ...
    end

  end
  describe '#comments' do
    # ...
  end
end

# good
describe Article do
  describe '#summary' do
    context 'when there is a summary' do
      # ...
    end

    context 'when there is no summary' do
      # ...
    end
  end

  describe '#comments' do
    # ...
  end
end
```

Empty Line After `let`

Leave one empty line after `let`, `subject`, and `before`/`after` blocks.

```ruby
# bad
describe Article do
  subject { FactoryBot.create(:some_article) }
  describe '#summary' do
    # ...
  end
end

# good
describe Article do
  subject { FactoryBot.create(:some_article) }

  describe '#summary' do
    # ...
  end
end
```

Let Grouping

Only group `let`, `subject` blocks and separate them from `before`/`after` blocks. It makes the code much more readable.

```ruby
# bad
describe Article do
  subject { FactoryBot.create(:some_article) }
  let(:user) { FactoryBot.create(:user) }
  before do
    # ...
  end
  after do
    # ...
  end
  describe '#summary' do
    # ...
  end
end

# good
describe Article do
  subject { FactoryBot.create(:some_article) }
  let(:user) { FactoryBot.create(:user) }

  before do
    # ...
  end

  after do
    # ...
  end

  describe '#summary' do
    # ...
  end
end
```

Empty Lines around Examples

Leave one empty line around `it`/`specify` blocks. This helps to separate the expectations from their conditional logic (contexts for instance).

```ruby
# bad
describe '#summary' do
  let(:item) { double('something') }

  it 'returns the summary' do
    # ...
  end
  it 'does something else' do
    # ...
  end
  it 'does another thing' do
    # ...
  end
end

# good
describe '#summary' do
  let(:item) { double('something') }

  it 'returns the summary' do
    # ...
  end

  it 'does something else' do
    # ...
  end

  it 'does another thing' do
    # ...
  end
end
```

Leading `subject`

When `subject` is used, it should be the first declaration in the example group.

```ruby
# bad
describe Article do
  before do
    # ...
  end

  let(:user) { FactoryBot.create(:user) }
  subject { FactoryBot.create(:some_article) }

  describe '#summary' do
    # ...
  end
end

# good
describe Article do
  subject { FactoryBot.create(:some_article) }
  let(:user) { FactoryBot.create(:user) }

  before do
    # ...
  end

  describe '#summary' do
    # ...
  end
end
```

Example Group Structure

Use Contexts

Use contexts to make the tests clear, well organized, and easy to read.

```ruby
# bad
it 'has 200 status code if logged in' do
  expect(response).to respond_with 200
end

it 'has 401 status code if not logged in' do
  expect(response).to respond_with 401
end

# good
context 'when logged in' do
  it { is_expected.to respond_with 200 }
end

context 'when logged out' do
  it { is_expected.to respond_with 401 }
end
```

Context Cases

`context` blocks should pretty much always have an opposite negative case. It is a code smell if there is a single context (without a matching negative case), and this code needs refactoring, or may have no purpose.

```ruby
# bad - needs refactoring
describe '#attributes' do
  context 'the returned hash' do
    it 'includes the display name' do
      # ...
    end

    it 'includes the creation time' do
      # ...
    end
  end
end

# bad - the negative case needs to be tested, but isn't
describe '#attributes' do
  context 'when display name is present' do
    before do
      subject.display_name = 'something'
    end

    it 'includes the display name' do
      # ...
    end
  end
```

```ruby
end

# good
describe '#attributes' do
  subject { FactoryBot.create(:article) }

  specify do
    expect(subject.attributes).to include subject.display_name
    expect(subject.attributes).to include subject.created_at
  end
end

describe '#attributes' do
  context 'when display name is present' do
    before do
      subject.display_name = 'something'
    end

    it 'includes the display name' do
      # ...
    end
  end

  context 'when display name is not present' do
    before do
      subject.display_name = nil
    end

    it 'does not include the display name' do
      # ...
    end
  end
end
```

`let` Blocks

Use `let` and `let!` for data that is used across several examples in an example group. Use `let!` to define variables even if they are not referenced in some of the examples, e.g. when testing balancing negative cases. Do not overuse `let`s for primitive data, find the balance between frequency of use and complexity of the definition.

```
# bad
it 'finds shortest path' do
  tree = Tree.new(1 => 2, 2 => 3, 2 => 6, 3 => 4, 4 => 5, 5 => 6)
  expect(dijkstra.shortest_path(tree, from: 1, to: 6)).to eq([1, 2, 6])
end

it 'finds longest path' do
  tree = Tree.new(1 => 2, 2 => 3, 2 => 6, 3 => 4, 4 => 5, 5 => 6)
  expect(dijkstra.longest_path(tree, from: 1, to: 6)).to eq([1, 2, 3, 4, 5, 6])
end

# good
let(:tree) { Tree.new(1 => 2, 2 => 3, 2 => 6, 3 => 4, 4 => 5, 5 => 6) }

it 'finds shortest path' do
  expect(dijkstra.shortest_path(tree, from: 1, to: 6)).to eq([1, 2, 6])
end

it 'finds longest path' do
  expect(dijkstra.longest_path(tree, from: 1, to: 6)).to eq([1, 2, 3, 4, 5, 6])
end
```

Instance Variables

Use `let` definitions instead of instance variables.

```
# bad
before { @name = 'John Wayne' }

it 'reverses a name' do
  expect(reverser.reverse(@name)).to eq('enyaW nhoJ')
end

# good
let(:name) { 'John Wayne' }

it 'reverses a name' do
  expect(reverser.reverse(name)).to eq('enyaW nhoJ')
end
```

Shared Examples

Use shared examples to reduce code duplication.

```
# bad
describe 'GET /articles' do
```

```
  let(:article) { FactoryBot.create(:article, owner: owner) }

  before { page.driver.get '/articles' }

  context 'when user is the owner' do
    let(:user) { owner }

    it 'shows all owned articles' do
      expect(page.status_code).to be(200)
      contains_resource resource
    end
  end

  context 'when user is an admin' do
    let(:user) { FactoryBot.create(:user, :admin) }

    it 'shows all resources' do
      expect(page.status_code).to be(200)
      contains_resource resource
    end
  end
end

# good
describe 'GET /articles' do
  let(:article) { FactoryBot.create(:article, owner: owner) }

  before { page.driver.get '/articles' }

  shared_examples 'shows articles' do
    it 'shows all related articles' do
      expect(page.status_code).to be(200)
      contains_resource resource
    end
  end

  context 'when user is the owner' do
    let(:user) { owner }

    include_examples 'shows articles'
  end

  context 'when user is an admin' do
    let(:user) { FactoryBot.create(:user, :admin) }

    include_examples 'shows articles'
  end
end

# good
describe 'GET /devices' do
```

```
  let(:resource) { FactoryBot.create(:device, created_from: user) }

  it_behaves_like 'a listable resource'
  it_behaves_like 'a paginable resource'
  it_behaves_like 'a searchable resource'
  it_behaves_like 'a filterable list'
end
```

Redundant `before(:each)`

Don't specify `:each`/`:example` scope for `before`/`after`/`around` blocks, as it is the default. Prefer `:example` when explicitly indicating the scope.

```
# bad
describe '#summary' do
  before(:example) do
    # ...
  end

  # ...
end

# good
describe '#summary' do
  before do
    # ...
  end

  # ...
end
```

Ambiguous Hook Scope

Use `:context` instead of the ambiguous `:all` scope in `before`/`after` hooks.

```
# bad
describe '#summary' do
  before(:all) do
    # ...
  end

  # ...
end

# good
describe '#summary' do
  before(:context) do
    # ...
  end

  # ...
end
```

Avoid Hooks with `:context` Scope

Avoid using `before`/`after` with `:context` scope. Beware of the state leakage between the examples.

Example Structure

Expectation per Example

For examples two styles are considered acceptable. The first variant is separate example for each expectation, which comes with a cost of repeated context initialization. The second variant is multiple expectations per example with `aggregate_failures` tag set for a group or example. Use your best judgement in each case, and apply your strategy consistently.

```ruby
# good - one expectation per example
describe ArticlesController do
  #...

  describe 'GET new' do
    it 'assigns a new article' do
      get :new
      expect(assigns[:article]).to be_a(Article)
    end

    it 'renders the new article template' do
      get :new
      expect(response).to render_template :new
    end
  end
end

# good - multiple expectations with aggregated failures
describe ArticlesController do
  #...

  describe 'GET new', :aggregate_failures do
    it 'assigns new article and renders the new article template' do
      get :new
      expect(assigns[:article]).to be_a(Article)
      expect(response).to render_template :new
    end
  end

  # ...
end
```

Subject

When several tests relate to the same subject, use `subject` to reduce repetition.

```ruby
# bad
it { expect(hero.equipment).to be_heavy }
it { expect(hero.equipment).to include 'sword' }

# good
subject(:equipment) { hero.equipment }

it { expect(equipment).to be_heavy }
it { expect(equipment).to include 'sword' }
```

Named Subject

Use named `subject` when possible. Only use anonymous subject declaration when you don't reference it in any tests, e.g. when `is_expected` is used.

```ruby
# bad
describe Article do
  subject { FactoryBot.create(:article) }

  it 'is not published on creation' do
    expect(subject).not_to be_published
  end
end

# good
describe Article do
  subject { FactoryBot.create(:article) }

  it 'is not published on creation' do
    is_expected.not_to be_published
  end
end

# even better
describe Article do
  subject(:article) { FactoryBot.create(:article) }

  it 'is not published on creation' do
    expect(article).not_to be_published
  end
end
```

Subject Naming in Context

When you reassign subject with different attributes in different contexts, give different names to the subject, so it's easier to see what the actual subject represents.

```ruby
# bad
describe Article do
  context 'when there is an author' do
    subject(:article) { FactoryBot.create(:article, author: user) }

    it 'shows other articles by the same author' do
      expect(article.related_stories).to include(story1, story2)
    end
  end

  context 'when the author is anonymous' do
    subject(:article) { FactoryBot.create(:article, author: nil) }

    it 'matches stories by title' do
      expect(article.related_stories).to include(story3, story4)
    end
  end
end

# good
describe Article do
  context 'when article has an author' do
    subject(:article) { FactoryBot.create(:article, author: user) }

    it 'shows other articles by the same author' do
      expect(article.related_stories).to include(story1, story2)
    end
  end

  context 'when the author is anonymous' do
    subject(:guest_article) { FactoryBot.create(:article, author: nil) }

    it 'matches stories by title' do
      expect(guest_article.related_stories).to include(story3, story4)
    end
  end
end
```

Don’t Stub Subject

Don’t stub methods of the object under test, it’s a code smell and often indicates a bad design of the object itself.

```ruby
# bad
describe 'Article' do
  subject(:article) { Article.new }

  it 'indicates that the author is unknown' do
    allow(article).to receive(:author).and_return(nil)
    expect(article.description).to include('by an unknown author')
  end
end

# good - with correct subject initialization
describe 'Article' do
  subject(:article) { Article.new(author: nil) }

  it 'indicates that the author is unknown' do
    expect(article.description).to include('by an unknown author')
  end
end

# good - with better object design
describe 'Article' do
  subject(:presenter) { ArticlePresenter.new(article) }
  let(:article) { Article.new }

  it 'indicates that the author is unknown' do
    allow(article).to receive(:author).and_return(nil)
    expect(presenter.description).to include('by an unknown author')
  end
end
```

`it` and `specify`

Use `specify` if the example doesn’t have a description, use `it` for examples with descriptions. An exception is one-line example, where `it` is preferable. `specify` is also useful when the docstring does not read well off of `it`.

```ruby
# bad
it do
  # ...
end

specify 'it sends an email' do
  # ...
end

specify { is_expected.to be_truthy }

it '#do_something is deprecated' do
  ...
end

# good
specify do
  # ...
end

it 'sends an email' do
  # ...
end

it { is_expected.to be_truthy }

specify '#do_something is deprecated' do
  ...
end
```

`it` in Iterators

Do not write iterators to generate tests. When another developer adds a feature to one of the items in the iteration, they must then break it out into a separate test - they are forced to edit code that has nothing to do with their pull request.

```ruby
# bad
[:new, :show, :index].each do |action|
  it 'returns 200' do
    get action
    expect(response).to be_ok
  end
end

# good - more verbose, but better for the future development
describe 'GET new' do
  it 'returns 200' do
    get :new
    expect(response).to be_ok
  end
end

describe 'GET show' do
  it 'returns 200' do
    get :show
    expect(response).to be_ok
  end
end

describe 'GET index' do
  it 'returns 200' do
    get :index
    expect(response).to be_ok
  end
end
```

Incidental State

Avoid incidental state as much as possible.

```ruby
# bad
it 'publishes the article' do
  article.publish

  # Creating another shared Article test object above would cause this
  # test to break
  expect(Article.count).to eq(2)
end

# good
it 'publishes the article' do
  expect { article.publish }.to change(Article, :count).by(1)
end
```

DRY

Be careful not to focus on being 'DRY' by moving repeated expectations into a shared environment too early, as this can lead to brittle tests that rely too much on one another.

In general, it is best to start with doing everything directly in your `it` blocks even if it is duplication and then refactor your tests after you have them working to be a little more DRY. However, keep in mind that duplication in test suites is NOT frowned upon, in fact it is preferred if it provides easier understanding and reading of a test.

Factories

Use Factory Bot to create test data in integration tests. You should very rarely have to use `ModelName.create` within an integration spec. Do **not** use fixtures as they are not nearly as maintainable as factories.

```ruby
# bad
subject(:article) do
  Article.create(
    title: 'Piccolina',
    author: 'John Archer',
    published_at: '17 August 2172',
    approved: true
  )
end

# good
subject(:article) { FactoryBot.create(:article) }
```

NOTE When talking about unit tests the best practice would be to use neither fixtures nor factories. Put as much of your domain logic in libraries that can be tested without needing complex, time consuming setup with either factories or fixtures.

Needed Data

Do not load more data than needed to test your code.

```ruby
# good
RSpec.describe User do
  describe ".top" do
    subject { described_class.top(2) }

    before { FactoryBot.create_list(:user, 3) }

    it { is_expected.to have(2).items }
  end
end
```

Doubles

Prefer using verifying doubles over normal doubles.

Verifying doubles are a stricter alternative to normal doubles that provide guarantees, e.g. a failure will be triggered if an invalid method is being stubbed or a method is called with an invalid number of arguments.

In general, use doubles with more isolated/behavioral tests rather than with integration tests.

NOTE There is no justification for turning `verify_partial_doubles` configuration option off. That will significantly reduce the confidence in partial doubles.

```ruby
# good - verifying instance double
article = instance_double('Article')
allow(article).to receive(:author).and_return(nil)

presenter = described_class.new(article)
expect(presenter.title).to include('by an unknown author')


# good - verifying object double
article = object_double(Article.new, valid?: true)
expect(article.save).to be true


# good - verifying partial double
allow(Article).to receive(:find).with(5).and_return(article)


# good - verifying class double
notifier = class_double('Notifier')
expect(notifier).to receive(:notify).with('suspended as')
```

NOTE If you stub a method that could give a false-positive test result, you have gone too far.

Dealing with Time

Always use Timecop instead of stubbing anything on Time or Date.

```ruby
# bad
it 'offsets the time 2 days into the future' do
  current_time = Time.now
  allow(Time).to receive(:now).and_return(current_time)
  expect(subject.get_offset_time).to eq(current_time + 2.days)
end

# good
it 'offsets the time 2 days into the future' do
  Timecop.freeze(Time.now) do
    expect(subject.get_offset_time).to eq 2.days.from_now
  end
end
```

Stub HTTP Requests

Stub HTTP requests when the code is making them. Avoid hitting real external services.

Use webmock and VCR separately or together.

```ruby
# good
context 'with unauthorized access' do
  let(:uri) { 'http://api.lelylan.com/types' }

  before { stub_request(:get, uri).to_return(status: 401, body: fixture('401.json')) }

  it 'returns access denied' do
    page.driver.get uri
    expect(page).to have_content 'Access denied'
  end
end
```

Declare Constants

Do not explicitly declare classes, modules, or constants in example groups. Stub constants instead.

NOTE Constants, including classes and modules, when declared in a block scope, are defined in global namespace, and leak between examples.

```ruby
# bad
describe SomeClass do
  CONSTANT_HERE = 'I leak into global namespace'
end

# good
describe SomeClass do
  before do
    stub_const('CONSTANT_HERE', 'I only exist during this example')
  end
end

# bad
describe SomeClass do
  class FooClass < described_class
    def double_that
      some_base_method * 2
    end
  end

  it { expect(FooClass.new.double_that).to eq(4) }
end

# good - anonymous class, no constant needs to be defined
describe SomeClass do
  let(:foo_class) do
    Class.new(described_class) do
      def double_that
        some_base_method * 2
      end
    end
  end

  it { expect(foo_class.new.double_that).to eq(4) }
end

# good - constant is stubbed
describe SomeClass do
  before do
    foo_class = Class.new(described_class) do
                  def do_something
                  end
                end
    stub_const('FooClass', foo_class)
  end

  it { expect(FooClass.new.double_that).to eq(4) }
end
```

Implicit Block Expectations

Avoid using implicit block expectations.

```ruby
# bad
subject { -> { do_something } }
it { is_expected.to change(something).to(new_value) }

# good
it 'changes something to a new value' do
  expect { do_something }.to change(something).to(new_value)
end
```

Naming

Context Descriptions

Context descriptions should describe the conditions shared by all the examples within. Full example names (formed by concatenation of all nested block descriptions) should form a readable sentence.

A typical description will be an adjunct phrase starting with 'when', 'with', 'without', or similar words.

```ruby
# bad - 'Summary user is logged in no display name shows a placeholder'
describe 'Summary' do
 context 'user logged in' do
   context 'no display name' do
     it 'shows a placeholder' do
     end
   end
 end
end

# good - 'Summary when the user is logged in when the display name is blank shows a
placeholder'
describe 'Summary' do
 context 'when the user is logged in' do
   context 'when the display name is blank' do
     it 'shows a placeholder' do
     end
   end
 end
end
```

Example Descriptions

`it`/`specify` block descriptions should never end with a conditional. This is a code smell that the `it` most likely needs to be wrapped in a `context`.

```ruby
# bad
it 'returns the display name if it is present' do
  # ...
end

# good
context 'when display name is present' do
  it 'returns the display name' do
    # ...
  end
end

# This encourages the addition of negative test cases that might have
# been overlooked
context 'when display name is not present' do
  it 'returns nil' do
    # ...
  end
end
```

Keep Example Descriptions Short

Keep example description shorter than 60 characters.

Write the example that documents itself, and generates proper documentation format output.

```ruby
# bad
it 'rewrites "should not return something" as "does not return something"' do
  # ...
end

# good
it 'rewrites "should not return something"' do
  expect(rewrite('should not return something')).to
    eq 'does not return something'
end

# good - self-documenting
specify do
  expect(rewrite('should not return something')).to
    eq 'does not return something'
end
```

Example Group Naming

Prefix `describe` description with a hash for instance methods, with a dot for class methods.

Given the following exists:

```
class Article
  def summary
    # ...
  end

  def self.latest
    # ...
  end
end
```

```
# bad
describe Article do
  describe 'summary' do
    #...
  end

  describe 'latest' do
    #...
  end
end

# good
describe Article do
  describe '#summary' do
    #...
  end

  describe '.latest' do
    #...
  end
end
```

"Should" in Example Docstrings

Do not write 'should' or 'should not' in the beginning of your example docstrings. The descriptions represent actual functionality, not what might be happening. Use the third person in the present tense.

```ruby
# bad
it 'should return the summary' do
  # ...
end

# good
it 'returns the summary' do
  # ...
end
```

Describe the Methods

Be clear about what method you are describing. Use the Ruby documentation convention of `.` when referring to a class method's name and `#` when referring to an instance method's name.

```ruby
# bad
describe 'the authenticate method for User' do
  # ...
end

describe 'if the user is an admin' do
  # ...
end

# good
describe '.authenticate' do
  # ...
end

describe '#admin?' do
  # ...
end
```

Use `expect`

Always use the newer `expect` syntax.

Configure RSpec to only accept the new `expect` syntax.

```ruby
# bad
it 'creates a resource' do
  response.should respond_with_content_type(:json)
end

# good
it 'creates a resource' do
  expect(response).to respond_with_content_type(:json)
end
```

Matchers

Predicate Matchers

Use RSpec's predicate matcher methods when possible.

```ruby
# bad
it 'is published' do
  expect(subject.published?).to be true
end

# good
it 'is published' do
  expect(subject).to be_published
end
```

Built in Matchers

Use built-in matchers.

```ruby
# bad
it 'includes a title' do
  expect(article.title.include?('a lengthy title')).to be true
end

# good
it 'includes a title' do
  expect(article.title).to include 'a lengthy title'
end
```

`be` Matcher

Avoid using `be` matcher without arguments. It is too generic, as it pass on everything that is not `nil` or `false`. If that is the exact intend, use `be_truthy`. In all other cases it's better to specify what exactly is the expected value.

```
# bad
it 'has author' do
  expect(article.author).to be
end

# good
it 'has author' do
  expect(article.author).to be_truthy # same as the original
  expect(article.author).not_to be_nil # `be` is often used to check for non-nil value
  expect(article.author).to be_an(Author) # explicit check for the type of the value
end
```

Extract Common Expectation Parts into Matchers

Extract frequently used common logic from your examples into custom matchers.

```
# bad
it 'returns JSON with temperature in Celsius' do
  json = JSON.parse(response.body).with_indifferent_access
  expect(json[:celsius]).to eq 30
end

it 'returns JSON with temperature in Fahrenheit' do
  json = JSON.parse(response.body).with_indifferent_access
  expect(json[:fahrenheit]).to eq 86
end

# good
it 'returns JSON with temperature in Celsius' do
  expect(response).to include_json(celsius: 30)
end

it 'returns JSON with temperature in Fahrenheit' do
  expect(response).to include_json(fahrenheit: 86)
end
```

`any_instance_of`

Avoid using `allow_any_instance_of`/`expect_any_instance_of`. It might be an indication that the object under test is too complex, and is ambiguous when used with receive counts.

```
# bad
it 'has a name' do
  allow_any_instance_of(User).to receive(:name).and_return('Tweedledee')
  expect(account.name).to eq 'Tweedledee'
end

# good
let(:account) { Account.new(user) }

it 'has a name' do
  allow(user).to receive(:name).and_return('Tweedledee')
  expect(account.name).to eq 'Tweedledee'
end
```

Matcher Libraries

Use third-party matcher libraries that provide convenience helpers that will significantly simplify the examples, Shoulda Matchers are one worth mentioning.

```
# bad
describe '#title' do
  it 'is required' do
    article.title = nil
    article.valid?
    expect(article.errors[:title])
      .to contain_exactly('Article has no title')
    not
  end
end

# good
describe '#title' do
  it 'is required' do
    expect(article).to validate_presence_of(:title)
      .with_message('Article has no title')
  end
end
```

Rails: Integration

Test what you see. Deeply test your models and your application behaviour (integration tests). Do not add useless complexity testing controllers.

This is an open debate in the Ruby community and both sides have good arguments supporting their idea. People supporting the need of testing controllers will tell you that your integration tests don’t cover all use cases and that they are slow. Both are wrong. It is possible to cover all use cases and it’s possible to make them fast.

Rails: Views

View Directory Structure

The directory structure of the view specs `spec/views` matches the one in `app/views`. For example the specs for the views in `app/views/users` are placed in `spec/views/users`.

View Spec File Name

The naming convention for the view specs is adding `_spec.rb` to the view name, for example the view `_form.html.erb` has a corresponding spec `_form.html.erb_spec.rb`.

View Outer `describe`

The outer `describe` block uses the path to the view without the `app/views` part. This is used by the `render` method when it is called without arguments.

```ruby
# spec/views/articles/new.html.erb_spec.rb
describe 'articles/new.html.erb' do
  # ...
end
```

View Mock Models

Always mock the models in the view specs. The purpose of the view is only to display information.

View `assign`

The method `assign` supplies the instance variables which the view uses and are supplied by the controller.

```
# spec/views/articles/edit.html.erb_spec.rb
describe 'articles/edit.html.erb' do
  it 'renders the form for a new article creation' do
    assign(:article, double(Article).as_null_object)
    render
    expect(rendered).to have_selector('form',
      method: 'post',
      action: articles_path
    ) do |form|
      expect(form).to have_selector('input', type: 'submit')
    end
  end
end
```

Capybara Negative Selectors

Prefer capybara negative selectors over `to_not` with positive ones.

```
# bad
expect(page).to_not have_selector('input', type: 'submit')
expect(page).to_not have_xpath('tr')

# good
expect(page).to have_no_selector('input', type: 'submit')
expect(page).to have_no_xpath('tr')
```

View Helper Stub

When a view uses helper methods, these methods need to be stubbed. Stubbing the helper methods is done on the `template` object:

```
# app/helpers/articles_helper.rb
class ArticlesHelper
  def formatted_date(date)
    # ...
  end
end
```

```
# app/views/articles/show.html.erb
<%= 'Published at: #{formatted_date(@article.published_at)}' %>
```

```
# spec/views/articles/show.html.erb_spec.rb
describe 'articles/show.html.erb' do
  it 'displays the formatted date of article publishing' do
    article = double(Article, published_at: Date.new(2012, 01, 01))
    assign(:article, article)

    allow(template).to_receive(:formatted_date).with(article.published_at).and_return
('01.01.2012')

    render
    expect(rendered).to have_content('Published at: 01.01.2012')
  end
end
```

View Helpers

The helpers specs are separated from the view specs in the `spec/helpers` directory.

Rails: Controllers

Controller Models

Mock the models and stub their methods. Testing the controller should not depend on the model creation.

Controller Behaviour

Test only the behaviour the controller should be responsible about:

- Execution of particular methods
- Data returned from the action - assigns, etc.
- Result from the action - template render, redirect, etc.

```ruby
# Example of a commonly used controller spec
# spec/controllers/articles_controller_spec.rb
# We are interested only in the actions the controller should perform
# So we are mocking the model creation and stubbing its methods
# And we concentrate only on the things the controller should do

describe ArticlesController do
  # The model will be used in the specs for all methods of the controller
  let(:article) { double(Article) }

  describe 'POST create' do
    before { allow(Article).to receive(:new).and_return(article) }

    it 'creates a new article with the given attributes' do
      expect(Article).to receive(:new).with(title: 'The New Article Title').
and_return(article)
      post :create, message: { title: 'The New Article Title' }
    end

    it 'saves the article' do
      expect(article).to receive(:save)
      post :create
    end

    it 'redirects to the Articles index' do
      allow(article).to receive(:save)
      post :create
      expect(response).to redirect_to(action: 'index')
    end
  end
end
```

Controller Contexts

Use context when the controller action has different behaviour depending on the received params.

```ruby
# A classic example for use of contexts in a controller spec is creation or update
when the object saves successfully or not.

describe ArticlesController do
  let(:article) { double(Article) }

  describe 'POST create' do
    before { allow(Article).to receive(:new).and_return(article) }

    it 'creates a new article with the given attributes' do
      expect(Article).to receive(:new).with(title: 'The New Article Title').
and_return(article)
```

```
      post :create, article: { title: 'The New Article Title' }
    end

    it 'saves the article' do
      expect(article).to receive(:save)
      post :create
    end

    context 'when the article saves successfully' do
      before do
        allow(article).to receive(:save).and_return(true)
      end

      it 'sets a flash[:notice] message' do
        post :create
        expect(flash[:notice]).to eq('The article was saved successfully.')
      end

      it 'redirects to the Articles index' do
        post :create
        expect(response).to redirect_to(action: 'index')
      end
    end

    context 'when the article fails to save' do
      before do
        allow(article).to receive(:save).and_return(false)
      end

      it 'assigns @article' do
        post :create
        expect(assigns[:article]).to eq(article)
      end

      it "re-renders the 'new' template" do
        post :create
        expect(response).to render_template('new')
      end
    end
  end
end
```

Rails: Models

Model Mocks

Do not mock the models in their own specs.

Model Objects

Use `FactoryBot.create` to make real objects, or just use a new (unsaved) instance with `subject`.

```ruby
describe Article do
  let(:article) { FactoryBot.create(:article) }

  # Currently, 'subject' is the same as 'Article.new'
  it 'is an instance of Article' do
    expect(subject).to be_an Article
  end

  it 'is not persisted' do
    expect(subject).to_not be_persisted
  end
end
```

Model Mock Associations

It is acceptable to mock other models or child objects.

Avoid Duplication in Model Tests

Create the model for all examples in the spec to avoid duplication.

```ruby
describe Article do
  let(:article) { FactoryBot.create(:article) }
end
```

Check Model Validity

Add an example ensuring that the model created with `FactoryBot.create` is valid.

```ruby
describe Article do
  it 'is valid with valid attributes' do
    expect(article).to be_valid
  end
end
```

Model Validations

When testing validations, use `expect(model.errors[:attribute].size).to eq(x)` to specify the attribute which should be validated. Using `be_valid` does not guarantee that the problem is in the intended attribute.

```ruby
# bad
describe '#title' do
  it 'is required' do
    article.title = nil
    expect(article).to_not be_valid
  end
end

# preferred
describe '#title' do
  it 'is required' do
    article.title = nil
    article.valid?
    expect(article.errors[:title].size).to eq(1)
  end
end
```

Separate Example Group for Attribute Validations

Add a separate `describe` for each attribute which has validations.

```ruby
describe '#title' do
  it 'is required' do
    article.title = nil
    article.valid?
    expect(article.errors[:title].size).to eq(1)
  end
end

describe '#name' do
  it 'is required' do
    article.name = nil
    article.valid?
    expect(article.errors[:name].size).to eq(1)
  end
end
```

Naming Another Object

When testing uniqueness of a model attribute, name the other object `another_object`.

```
describe Article do
  describe '#title' do
    it 'is unique' do
      another_article = FactoryBot.create(:article, title: article.title)
      article.valid?
      expect(article.errors[:title].size).to eq(1)
    end
  end
end
```

Rails: Mailers

Mailer Mock Model

The model in the mailer spec should be mocked. The mailer should not depend on the model creation.

Mailer Expectations

The mailer spec should verify that:

- the subject is correct
- the sender e-mail is correct
- the e-mail is sent to the correct recipient
- the e-mail contains the required information

```
describe SubscriberMailer do
  let(:subscriber) { double(Subscription, email: 'johndoe@test.com', name: 'John Doe')
}

  describe 'successful registration email' do
    subject { SubscriptionMailer.successful_registration_email(subscriber) }

    its(:subject) { should == 'Successful Registration!' }
    its(:from) { should == ['info@your_site.com'] }
    its(:to) { should == [subscriber.email] }

    it 'contains the subscriber name' do
      expect(subject.body.encoded).to match(subscriber.name)
    end
  end
end
```

Recommendations

Correct Setup

Correctly set up RSpec configuration globally (`~/.rspec`), per project (`.rspec`), and in project override file that is supposed to be kept out of version control (`.rspec-local`). Use `rspec --init` to generate `.rspec` and `spec/spec_helper.rb` files.

```
# .rspec
--color
--require spec_helper

# .rspec-local
--profile 2
```

Related Guides

- Ruby Style Guide
- Rails Style Guide
- Minitest Style Guide

Contributing

Nothing written in this guide is set in stone. Everyone is welcome to contribute, so that we could ultimately create a resource that will be beneficial to the entire Ruby community.

Feel free to open tickets or send pull requests with improvements. Thanks in advance for your help!

You can also support the project (and RuboCop) with financial contributions via Patreon.

How to Contribute?

It's easy, just follow the contribution guidelines below:

- Fork the project on GitHub
- Make your feature addition or bug fix in a feature branch
- Include a good description of your changes
- Push your feature branch to GitHub
- Send a Pull Request

License

Credit

Inspiration was taken from the following:

HowAboutWe's RSpec style guide

Community Rails style guide

This guide was maintained by ReachLocal for a long while.

This guide includes material originally present in BetterSpecs (newer site older site), sponsored by Lelylan and maintained by Andrea Reginato and many others for a long while.