Extracted from:

Rails 5 Test Prescriptions

Build a Healthy Codebase

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rails 5 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

Edited by Katharine Dvorak

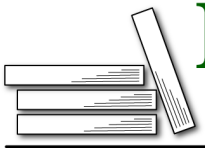
Rails 5 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

The Pragmatic Bookshelf

Raleigh, North Carolina



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
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Candace Cunningham
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: 978-1-68050-250-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

Where to Start?

"Where do I begin testing?" is one of the most common questions people have when they start with TDD. Traditionally, my answer is a somewhat glib "start anywhere." While true, this is less than helpful.

A good option for starting a TDD cycle is to specify the initialization state of the objects or methods under test. Another is the "happy path"—a single representative example of the error-free version of the algorithm. Which starting point you choose depends on how complicated the feature is. In this case it's sufficiently complex that I'll start with the initial state and move to the happy path. As a rule of thumb, if it takes more than a couple of steps to define an instance of the application, I'll start with initialization only.

Prescription 3	Initializing objects is a good starting place for a TDD process. Another good approach is to use the test to design what you want a successful interaction of the feature to look like.

This application is made up of projects and tasks. A newly created project would have no tasks. What can you say about that brand-new project?

If there are no outstanding tasks, then there's nothing left to do. A project with nothing left to do is done. The initial state, then, is a project with no tasks, and by that logic, the project is done. That's not an inevitable decision; you could specify that a project with no tasks is in some kind of empty state.

You don't have any infrastructure in place yet, so you need to create the test file yourself—we're deliberately not using Rails generators right now. We're using RSpec, so the spec goes in the spec directory using a file name that is parallel to the application code in the app directory. We think this is a test of a project model, which would be in app/models/project.rb, so put the spec in spec/models/project_spec.rb. We're making very small design decisions here, and so far these decisions are consistent with Rails conventions.

Here's your spec of a project's initial state:

basics/01/spec/models/project_spec.rb

```
Line 1  require "rails_helper"
    2
    3  RSpec.describe Project do
    4    it "considers a project with no tasks to be done" do
    5      project = Project.new
    6      expect(project.done?).to be_truthy
    7    end
    8
    9  end
```

Let's talk about this spec at two levels: the logistics of the code in RSpec and what this test is doing for you in your TDD process.

Compared to other testing libraries, RSpec shifts the tone from an "assertion," potentially implying already-implemented behavior, to an "expectation," implying future behavior. The RSpec version, arguably, is easier to read than the Minitest version (though some strenuously dispute this). Later in this chapter I'll cover some other tricks RSpec uses to make expectations read like natural language.

The project_spec.rb file uses four basic RSpec and Rails features:

- It requires rails_helper.
- It defines a test suite with RSpec.describe.
- It creates an RSpec example with it.
- It specifies a particular state with expect.

On the first line, the file rails_helper, which contains Rails-related setup common to all tests, is required. (You'll peek into that file in the next chapter, when I talk about more Rails-specific test features.) The rails_helper file, in turn, requires a file named spec_helper, which contains non-Rails RSpec setup.

The RSpec.describe method is used on line 3. In RSpec, the describe method defines a suite of specs that can share a common setup. The first argument to describe is either a class name or a string. The first argument documents what the test suite is supposed to cover. You can then pass an optional number of metadata arguments, of which there are none at the moment. The metadata is used to specify additional behavior for the spec. Finally, describe expects a block, which contains the test suite itself.

As you'll see in a little bit, describe calls can be nested. By convention, the outermost call typically has the name of the class under test. The outermost describe call must be invoked as RSpec.describe, since that call starts outside of RSpec's control. Nested calls can use just plain describe, since RSpec manages those calls internally.

The actual spec is defined with the it method, which takes an optional string argument that documents the spec, an optional amount of metadata, and then a block that is the body of the spec. The string argument is not used internally to identify the spec—you can have multiple specs with the same description string. Again, the metadata is used to adjust RSpec's behavior.

RSpec also defines specify as an alias for it. Normally, you'd use it when the method takes a string argument to give the spec a readable natural-language name. (Historically the string argument started with "should," so the name

would be something like “it should be valid,” but that documentation pattern is no longer considered a good practice.) For single-line tests in which a string description is unnecessary, you can use specify to make the single line read more clearly, such as this:

```
specify { expect(user.name).to eq("fred") }
```

On line 6 you write your first testable expectation about the code:

```
expect(project.done?).to be_truthy
```

The general form of an RSpec expectation is expect(actual_value).to(matcher), with the parentheses around the matcher often omitted in practice. In this case, the expectation would more formally read, expect(project.done?).to(be_truthy). A *matcher* is an RSpec object that takes a value and determines if it matches expectations based on some set of logic. In this example, the matcher is be_truthy.

Let’s trace through what RSpec does with this expectation. First is the expect call itself, expect(project.done?). RSpec defines the expect method, which takes in any object as an argument and returns a special RSpec proxy object called an ExpectationTarget.

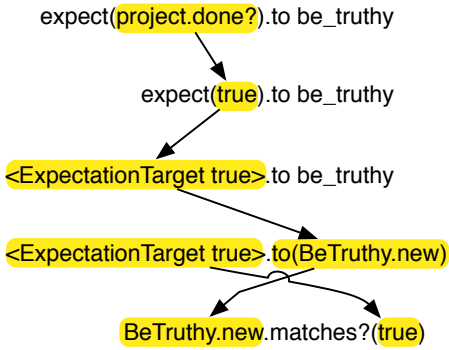
The ExpectationTarget holds on to the object that was the argument to expect, and itself responds to two messages: to and not_to. (Okay, technically three messages, since to_not exists as an alias.) Both to and not_to are ordinary Ruby methods that expect a single argument, which needs to be an RSpec matcher. There’s nothing special about an RSpec matcher; at base it’s just an object that responds to a matches? method. There are several predefined matchers and you can write your own.

In this case, be_truthy is a method defined by RSpec to return the BeTruthy matcher. You could get the same behavior with

```
expect(project.done?).to(RSpec::BuiltIn::BeTruthy.new)
```

but you probably would agree that the idiomatic version is easier to read.

The ExpectationTarget is now holding on to two objects: the object being matched (in this case, project.done?) and the matcher (be_truthy). When the spec is executed, RSpec calls the matches? method on the matcher, with the object being matched as an argument. If the expectation uses to, then the expectation passes if matches? is true. If the expectation uses not_to, then it checks for a does_not_match? method in the matcher. If there is no such method it falls back to passing if matches? is false. This is shown in the following diagram.



From an RSpec perspective, you're creating an object and asserting an initial condition. What are you doing from a TDD perspective, and why is this useful?

Small as it might seem, you've performed a little bit of design. You're starting to define the way parts of your system communicate with each other, and the tests ensure the visibility of important information in your design.

This small test makes three claims about your program:

- There is a class called `Project`.
- You can query instances of that class as to whether they are done.
- A brand-new instance of `Project` qualifies as done.

This last assertion isn't inevitable—you could say that you aren't done unless there is at least one completed task, but that's a choice you're making in the application's business logic.

RSpec Predefined Matchers

Before you run the tests, let's take a quick look at RSpec's basic matchers. RSpec predefines a number of matchers. What follows is a list of the most useful ones; you can find a full list online.[2]

```
expect(array).to all(matcher)
expect(actual).to be > expected     # (also works with <, >=, <=, and ==)
expect(actual).to be_a(type)
expect(actual).to be_truthy
expect(actual).to be_falsy
expect(actual).to be_nil
expect(actual).to be_between(min, max)
expect(actual).to be_within(delta).of(expected)
expect { block }.to change(receiver, message, &block)
expect(actual).to contain_exactly(expected)
expect(range).to cover(actual_value)
```

2. https://relishapp.com/rspec/rspec-expectations/v/3-7/docs/built-in-matchers

```
expect(actual).to eq(expected)
expect(actual).to exist
expect(actual).to have_attributes(key/value pairs)
expect(actual).to include(*expected)
expect(actual).to match(regex)
expect { block }.to output(value).to_stdout   # also to_stderr
expect { block }.to raise_error(exception)
expect(actual).to satisfy { block }
```

Most of these mean what they appear to say. Some elaborations:

- The all matcher takes a different matcher as an argument and passes if all elements of the array pass that internal matcher, as in expect([1, 2, 3]).to all(be_truthy).
- The change matcher takes a block argument that passes if the value of receiver.message changes when the block is evaluated.
- The contain_exactly matcher is true if the expected array and the actual array contain the same elements, regardless of order.
- The satisfy matcher passes if the block evaluates to true.
- The matchers that take block arguments, output and raise_error, are for specifying a side effect of the block's execution—that it raises an error or that it changes a different value—rather than the state of a particular object.

Any of these matchers except raise_error can be negated by using not_to instead of using to.

RSpec allows you to compose matchers to express compound behavior, and most of these matchers have alternate forms that allow them to read better when composed. Composing matchers allows you to specify, for example, multiple array values in a single statement and get useful error messages.

Here is a contrived example:

```
expect(["cheese", "burger"]).to contain_exactly(
    a_string_matching(/ch/), a_string_matching(/urg/))
```

In this case a_string_matching is an alias for match, and the arguments to contain_exactly are themselves matchers that must match individual elements of the array to allow the entire compound matcher to pass.