Extracted from:

Rails Test Prescriptions

Keeping Your Application Healthy

The Pragmatic Programmers

Rails Test Prescriptions

Keeping Your Application Healthy

Noel Rappin

Edited by Colleen Toporek

Chapter 3

Writing Your First Tests

You have a problem. You are the team leader for a development team that is distributed across multiple locations. As an agile development team, your project has a daily stand-up meeting, sometimes called a *scrum*, where everybody briefly describes what they did yesterday, what they plan on doing today, and if anything is blocking them from getting their work done.

However, since your team is geographically distributed, you need to do these scrums via email. That's not the worst thing ever, but it does lead to annoying email threads, and I think we can all do better with a little web application magic. Let's create an application called Huddle, which will support entering and viewing these daily status messages.

Since you are a Rails developer who wants to use test-driven methods, the first thing you should ask is, "What do I test?" Test-driven developers start an application by writing tests. In that spirit, we're going to initiate our tour of Rails testing by writing lots of tests. Specifically, we're going to walk through the first few test-driven feature cycles of the Huddle application to give you the feel of Test-Driven Development (TDD) using Rails.

We'll use a hands-on approach and walk through the specifics of how to write your first tests. We'll talk about how the practice of working "test first" improves development, but more importantly, we'll show what working in a test-driven style looks like. This chapter uses the testing tools that are available in core Rails and will be limited to common Rails tasks such as creating and submitting a web form. At the end of this chapter, you should have a good sense of how TDD development

A Word About Best Practices

There's a tension in this section between making the introduction to Rails testing as simple and clear as possible and presenting the tests using what I would consider to be best practices. In particular, many of my regular testing practices depend on third-party tools that we're not going to cover in this walk-through.

In this chapter, I decided to focus on making testing as easy as possible to explain while still using good coding practice, and I included some discussion of where improvements might come. We'll go over coding style and practice considerations again later in the book.

works in Rails, and you'll be ready to explore the third-party tools and more detailed topics in the rest of the book.

Appendix A, on page 323, contains the steps for creating the skeleton application we're starting with—including the initial setup, creation of Rails scaffolds, addition of Devise for user authentication, and other things that are necessary to the application but beside the point for our tutorial. If you'd like to start at the same place, the code samples for this chapter are available for download at http://www.pragprog.com/titles/nrtest/source_code. The code for this application was written and tested against Rails 3.0.[1]

We're going to do this in a reasonably strict test-driven style, meaning no new logic will be added to the application except in response to a failing test. We'll be a little more lenient with view code. We're assuming a basic understanding of standard Rails concepts; in other words, you don't need to be told what a controller is. For the moment, we're also going to limit ourselves to test tools provided by core Rails. Later in the book, we'll spend a lot of time covering third-party tools, especially in Chapter 11, *Write Cleaner Tests with Shoulda and Contexts*, on page 171 and Chapter 12, *RSpec*, on page 188. But in the name of keeping it simple, we'll start with vanilla core Rails.

1. Significant differences with Rails 2.3.*x* will be noted.

3.1 The First Test-First

The first question to ask is, “What do I test?” The answer comes from your requirements. Without some sense of what your program should be doing, it’s hard to write tests that describe that behavior in code.

The form and formality of your requirements will depend on the needs of your project. In this case, you are your own client, and it’s kind of a small project, and we don’t have space in this book for military-level precision. So, the informal list of the first three stories in the application looks something like this:

- A user is part of a project. A user can enter his scrum status for that project.
- For the purpose of adding a testable constraint, let’s say the user’s status report has yesterday’s status and today’s expected work, and the user must include text in at least one of these items.
- Members of the project can see a timeline of status reports. This one will get covered in Chapter 4, *TDD, Rails Style*, on page 63.

Over the rest of this tutorial, we’ll go after these stories one by one. Any time we add or change the logic of the application, we’ll write a test. The exact starting point of the first test is not important (although it’s helpful to have at least some sense of where you are going); you can start with any requirement or feature in the program that can be objectively specified.

Our starting point for Huddle is the need to have a status report that is created as part of a project. The report should have all its values, including the date, set correctly. Because I think the code for this feature might be in the StatusReportsController, I’m going to put the test for this feature in test/functional/status_reports_controller_test.rb.

```ruby
test "creation of status report with data" do
  assert_difference('StatusReport.count', 1) do
    post :create, :status_report => {
      :project_id => projects(:one).to_param,
      :user_id => users(:one).to_param,
      :yesterday => "I did stuff",
      :today => "I'll do stuff"}
  end
  actual = assigns(:status_report)
  assert_equal(projects(:one).id, actual.project.id)
  assert_equal(users(:one).id, actual.user.id)
  assert_equal(Date.today.to_s(:db), actual.status_date.to_s(:db))
  assert_redirected_to status_report_path(actual)
end
```

Let's walk through this test in detail.

Line 3 simulates a post call to the create action of the StatusReportsController. The second argument to this call simulates the URL parameters of the call—effectively, you are setting up the params hash that will be used in the action. As part of that hash, the call references users(:one), which is a *fixture*, or set of known sample data that can be used in testing. This particular fixture set was created in Appendix A, on page 323, and it defines the data object accessed as users(:one). Section 2.7, *More Info: Getting Data into the Test*, on page 40 has more detail on fixtures.

Going back to the test itself, the block that starts in line 2 and ends in line 8 uses the assert_difference() method to assert that there is one more StatusReport object in the database at the end of the block than at the beginning. More plainly, the method is asserting that a new StatusReport instance has been created.

Line 9 uses the Rails test framework assigns() method, allowing access to instance variables set in the controller being tested—in this case, the controller variable @status_report, which should be the newly created instance. You don't need the @ symbol in the argument to assigns().

Starting with line 10, there are three lines asserting that a project, user, and status date are added to the newly created object.[2] Line 13 asserts that the result of the controller call is a redirect to the show page of the newly created StatusReport.

Although people will certainly quibble with the style and structure of this test, it is a basic, straightforward test of the desired functionality. This is the maximum amount of complexity that I'm comfortable having in a single test. In some cases, the amount of data or validation needed in a test suggests the need to refactor some of the complexity into *setup methods* or *custom assertion methods*.

Rather than start with a controller test, I could start by testing the model behavior. The model test is probably closer to the code that will be written, since good Rails style places complexity in the model. However, I sometimes find that it is easier to specify the desired result when I start testing via the controller. Another option would be to start with an integration or Cucumber-based acceptance test (described in more detail in Chapter 15, *Acceptance Testing with Cucumber*, on page 237).

2. If this test is run at just the right moment before midnight, 12 will fail because the date has changed during the running of the code. Section 6.11, *Managing Date and Time Data*, on page 97 discusses working around this problem in more detail.

We're testing status_date because we know new code will be needed to add that attribute to the object, and we're testing the existence of the project and user objects because the requirements need relationships to be set up between the models. We're not testing the today and yesterday texts because that's part of core ActiveRecord—we could test it, but it would be redundant. Redundancy is not always bad in testing, but right now it's unnecessary.

I often use a testing style that limits each individual test to a single assertion and might therefore separate this test into four different tests sharing a common setup. The advantage of this one-assertion-per-test style is that each assertion is able to pass or fail separately. As written, the first failure prevents the rest of the tests from running. Although it's a good point that assertions should be independent, in this case it's easier to follow the intent of the test when similar assertions are grouped. Also, single assertion tests are easier to write with a little help from third-party tools. In Section 11.7, *Single-Line Test Tools*, on page 185, we'll see some tools that make it easier to write single-assertion tests.

When we run the tests, we get an error. The stack trace for the error looks like this:

```
  1) Error:
test_creation_of_status_report_with_data(StatusReportsControllerTest):
ArgumentError: wrong number of arguments (1 for 0)
    /test/functional/status_reports_controller_test.rb:58:in `to_s'
    /test/functional/status_reports_controller_test.rb:58:in
     `test_creation_of_status_report_with_data'
```

The line with the error is assert_equal(Date.today.to_s(:db), actual.status_date.to_s(:db)), and strictly speaking, the error message says that to_s, which converts the object to a string, is being called with the wrong number of arguments: (1 for 0), which means the method was called with one argument but expected zero.

This error message is technically true but misleading. The real error is that actual.status_date is nil and not Date.today. That error manifests itself as a "wrong number of arguments" because the test converts both dates to strings. The method to_s() takes no arguments for most classes, but Rails ActiveSupport overrides the method for Date with an optional format argument. Since our test results in a nil value instead of a Date, the extra argument causes an error.[3]

3. Why convert to strings, you ask? Because you get much more readable error messages if the values are both strings.

Also, notice that the user and project parts of the test already pass. This is a Rails feature. With the use of user:references in the script/generate command line (the exact setup commands are listed in Appendix A, on page 323), Rails automatically adds the belongs_to association to the StatusReport class. As we'll see later, it doesn't add the relationship in the other direction.

Now let's make the test pass. The classic process says to do the simplest thing that could possibly work. It's a good idea to just make the immediate error or failure go away, even if we suspect there are further errors waiting in the test. Doing so keeps the test/code cycle short and prevents the code from getting unnecessarily complex.

To get past the test failure, add a line toward the beginning of the create() method in app/controllers/status_reports_controller.rb so that the method starts like so:

```
def create
  @status_report = StatusReport.new(params[:status_report])
  @status_report.status_date = Date.today         # ==> the new line
  ## the rest of the method as before
end
```

3.2 The First Refactor

We fixed the immediate problem, and the test passes. We now enter the refactoring step. There isn't much here to refactor, but we have one detail we can tweak: it's better not to set the status_date in the controller. Good Rails practice moves complexity from controllers to models where possible. For one thing, placing code in the models tends to decrease duplication where functionality is used by multiple controller actions. For another, code in the model is easier to test.

Ordinarily, we would not be writing tests during refactoring, just using existing tests to verify that behavior hasn't changed. However, when moving code from one layer, the controller, to another, the model, it helps to create tests in the new class. Especially here, because our new behavior will be slightly different, we want the status_date to be automatically set whenever the report is saved.

The unit test goes in test/unit/status_report_test.rb:

```
test "saving a status report saves the status date" do
  actual = StatusReport.new
  actual.save
  assert_equal(Date.today.to_s, actual.status_date.to_s)
end
```

The test fails. As referenced in a previous footnote, in line 4 we're comparing literal string objects rather than the dates.

To pass the test, we add a before_save() callback to the StatusReport class:

```
class StatusReport < ActiveRecord::Base
  belongs_to :project
  belongs_to :user

  before_save :set_status_date

  def set_status_date
    self.status_date = Date.today
  end
end
```

Now the test passes. But there's one more thing to worry about—if the status_date has already been set before the report is saved, the original date should be used. As the code stands now, the status_date will change whenever the model is edited. In the TDD process, we force ourselves to make that code change by exposing the error with a test. Here's how, in test/unit/status_report_test.rb:

```
Line 1   test "saving a status report that has a date doesn't override" do
     2     actual = StatusReport.new(:status_date => 10.days.ago.to_date)
     3     actual.save
     4     actual.reload
     5     assert_equal(10.days.ago.to_date.to_s, actual.status_date.to_s)
     6   end
```

The to_date() methods in lines 2 and 5 are there to convert between 10.days.ago, which is a Ruby DateTime object, and the status_date, which is a Ruby Date object. Without that conversion, we will get an error because the string formats won't match in line 5.

The reload() call in line 4 forces ActiveRecord to re-retrieve the record from the database. ActiveRecord does not prevent a database record from having multiple live objects pointing to it. In this particular case, the controller creates a new instance from the database and saves that instance, without touching the actual variable created for the test. As a result, the database version has typecast the status_date to a Date when saving, but the live version in memory hasn't gotten that change.

In general, it's a good idea to reload any object being tested and saved. This is most commonly an issue in controller tests, where you might create an object during setup and then another object is created during the controller action that is backed by the same database record. In that case, the object you are holding on to in the test does not reflect changes made to the database during the controller action, leading to hours of fun as you try to figure out why your test is failing. Reloading will allow the object in your tests to see changes to the database made after the object was created.

One way to make the new test pass is this very slight change to the model:

Download huddle3/app/models/status_report.rb

```
def set_status_date
  self.status_date = Date.today if status_date.nil?
end
```

And now the scary part: removing the status-changing line from the controller and making sure that the tests pass again. This involves removing the line of code that we just added to the controller a couple of seconds ago.

It just takes a second to remove the line, and then we can rerun rake to verify that the tests still pass.

3.3 More Validations

While we're looking at the status report model, there is another one of our original three requirements we can cover, namely, the requirement that a user must enter text in at least one of the yesterday and today boxes. Back in test/unit/status_report_test.rb:

Download huddle3/test/unit/status_report_test.rb

```
test "a report with both blank is not valid" do
  actual = StatusReport.new(:today => "", :yesterday => "")
  assert !actual.valid?
end
```

The simplest way to pass this test is by placing the following line of code in app/models/status_report.rb:

```
validates_presence_of :yesterday, :today
```

That's great! With that line of code in place, everything will be swell. Nothing can go wrong. (Cue ominous music.) Let's run rake:

```
  1) Failure:
test_saving_a_status_report_saves_the_status_date(StatusReportTest)
[/test/unit/status_report_test.rb:9]:
<"2009-08-26"> expected but was
<"">.
```

```
  2) Error:
   test_saving_with_a_date_doesn't_override(StatusReportTest):
ActiveRecord::RecordNotFound: Couldn't find StatusReport without an ID
    /test/unit/status_report_test.rb:17:in
   `test_saving_with_a_date_doesn't_override'
```

What? Well, you've probably figured it out, but adding the validation causes problems in other tests.[4] Specifically, status reports that were created by other tests without either text field being set are now failing their saves because they are invalid. This is admittedly annoying, because it's not really a regression in the code: the actual code in the browser probably still works fine. It's more that the shifting definition of what makes a valid StatusReport is now tripping up older tests that used insufficiently robust data.

Fixing the failing tests is straightforward. To fix the two tests in test/unit/status_report_test.rb, add the arguments (:today => "t", :yesterday => "y") to each StatusReport.new() method call, giving the following:

Download huddle3/test/unit/status_report_test.rb

```
test "saving a status report saves the status date" do
  actual = StatusReport.new(:today => "t", :yesterday => "y")
  actual.save
  assert_equal(Date.today.to_s, actual.status_date.to_s)
end

test "saving with a date doesn't override" do
  actual = StatusReport.new(:status_date => 10.days.ago.to_date,
      :today => "t", :yesterday => "y")
  actual.save
  actual.reload
  assert_equal(10.days.ago.to_date.to_s, actual.status_date.to_s)
end
```

This puts enough data in the report to make the test pass—we don't need to care what the data actually is. For Rails 2.*x*, a similar change needs to be made in test/functional/status_reports_controller_test.rb:

Download huddle/test/functional/status_reports_controller_test.rb

```
test "should create status_report" do
  assert_difference('StatusReport.count') do
    post :create, :status_report => {:today => "t", :yesterday => "y"}
  end
  assert_redirected_to status_report_path(assigns(:status_report))
end
```

4. In Rails 2.*x*, you also get a test failure in StatusReportTest for the test of the create action because of a difference in the behavior of the generated test. In Rails 3, that action is passed default values based on fixture data, so the validation works. In Rails 2, the generated test passes an empty hash to the controller.

And now we're back at all passing. This is, frankly, the kind of thing that causes people to develop an aversion to testing: sometimes it seems like a boatload of busywork to have to go back in and change all those older tests. And, well, it can be. There are a couple of ways you can minimize the annoyance and keep the benefits of working test-first.

One helpful technique is to keep a very tight loop between writing tests and writing code and to run the test suite frequently (ideally, we'd run it constantly using autotest or a similar continuous-test execution tool, Section 19.3, *Using Autotest*, on page 311). The tighter the loop and the fewer lines of code we write in each back-and-forth, the easier it is to find and track down these structural test problems.

Second, and more specific to these kinds of validation problems, using some kind of factory tool or common setup method to generate well-structured default data makes it much easier to keep data in sync with changing definitions of validity. Much more on that topic in Section 6.4, *Using Factories to Fix Fixtures*, on page 88.

Anyway, fixing the older data is a distraction: we have a larger problem. Remember, we wanted the status to be invalid only if *both* today and yesterday were blank. We need to write a couple of follow-up tests to confirm that we haven't overshot the mark. The tests go in test/unit/status_report_test.rb.

Download huddle3/test/unit/status_report_test.rb

```ruby
test "a report with yesterday blank is valid" do
  actual = StatusReport.new(:today => "today", :yesterday => "")
  assert actual.valid?
end

test "a report with today blank is valid" do
  actual = StatusReport.new(:today => "", :yesterday => "yesterday")
  assert actual.valid?
end
```

Oops.

```
  1) Failure:
test_a_test_with_today_blank_is_valid(StatusReportTest)
[/test/unit/status_report_test.rb:36]:
<false> is not true.

  2) Failure:
test_a_test_with_yesterday_blank_is_valid(StatusReportTest)
[/test/unit/status_report_test.rb:31]:
<false> is not true.
```

At this point, we want to move to a custom validation, because the validation functions provided by Rails won't quite get this right for us.

Replace the validation line in app/model/status_report.rb with the following call to plain validate() and the associated method:

Download huddle3/app/models/status_report.rb

```
validate :validate_has_at_least_one_status

def validate_has_at_least_one_status
  if today.blank? && yesterday.blank?
    errors[:base] << "Must have at least one status set"
  end
end
```

And we're back to passing.[5] This, by the way, is the first line of code we've seen in this chapter that is different for Rails 3 and Rails 2. The previous is for Rails 3. In Rails 2, the error is added with the method call errors.add_to_base().

Here are a couple of points on the question of what to test and when:

- The general situation here is very important. Always try to test a boundary from both sides. If you are testing that an administrator should see a certain link, you also need to test that a regular user can't see it. Your tests will give you an accurate picture of your application only if they cover the requirement boundaries from both sides.
- Although we don't need to test the Rails validation methods as such, we do need to verify the operational behavior that a model object in a certain state is invalid. In a strict TDD process, it's the test for validity that causes us to add the Rails validation method in the first place.
- Whether to go back and add a controller test to validate behavior for invalid objects is an open question. As a matter of course, we insert a generic test into our controller scaffold using mock objects to cover the general failure case (shown in detail in Chapter 7, *Using Mock Objects*, on page 103), which means we don't need to go back and test the controller behavior for each and every different possible kind of model failure, unless, of course, each specific failure actually dictates different controller behavior.

5. One early reviewer pointed out that this can, in fact, be done with the core Rails validations, namely, a pair of validates_presence_of() calls with the if option.

Now, this may seem like a lot of work because we've been going through every step in excruciating detail. In practice, though, each of these test cycles is very quick—in the five- to fifteen-minute range for relatively simple tests like these.

3.4 Security Now!

Let's take a look at Huddle's login and security models that use the Devise gem. Devise has its own set of tests, so we don't need to write tests for the basic behavior of login and logout. We do need to write tests to cover parts of the application-specific security model for who can see and edit what different things. Let's say that our authentication requirements are as follows:

1. Users must be logged in to view or create a status report.
2. Users must always have a current project chosen. Right now, any user can see and create a status report on any project. Assigning users to projects may or may not happen later. At the moment, we don't care.
3. Users can only edit their own reports. Again, there may or may not be admin functionality later; we'll cross that bridge when we get to it.

To enforce a Devise login globally throughout the app, we need to add the following inside the ApplicationController. In a slight break from normal procedure, we'll implement the forced login in the code first.

Download huddle3/app/controllers/application_controller.rb

```
before_filter :authenticate_user!
```

Why not do this test first? It's because most of the functionality is already tested by Devise and because the authentication model is super-basic and application-wide. If and when the login model gets more complex (if, for example, there were public reports that did not require a login), we'd start adding some tests.

Despite not adding any new tests, we suddenly have no shortage of failing tests just from adding the login requirement. Running rake, the unit tests pass, but the controller tests...well:

```
15 tests, 0 assertions, 0 failures, 15 errors
```