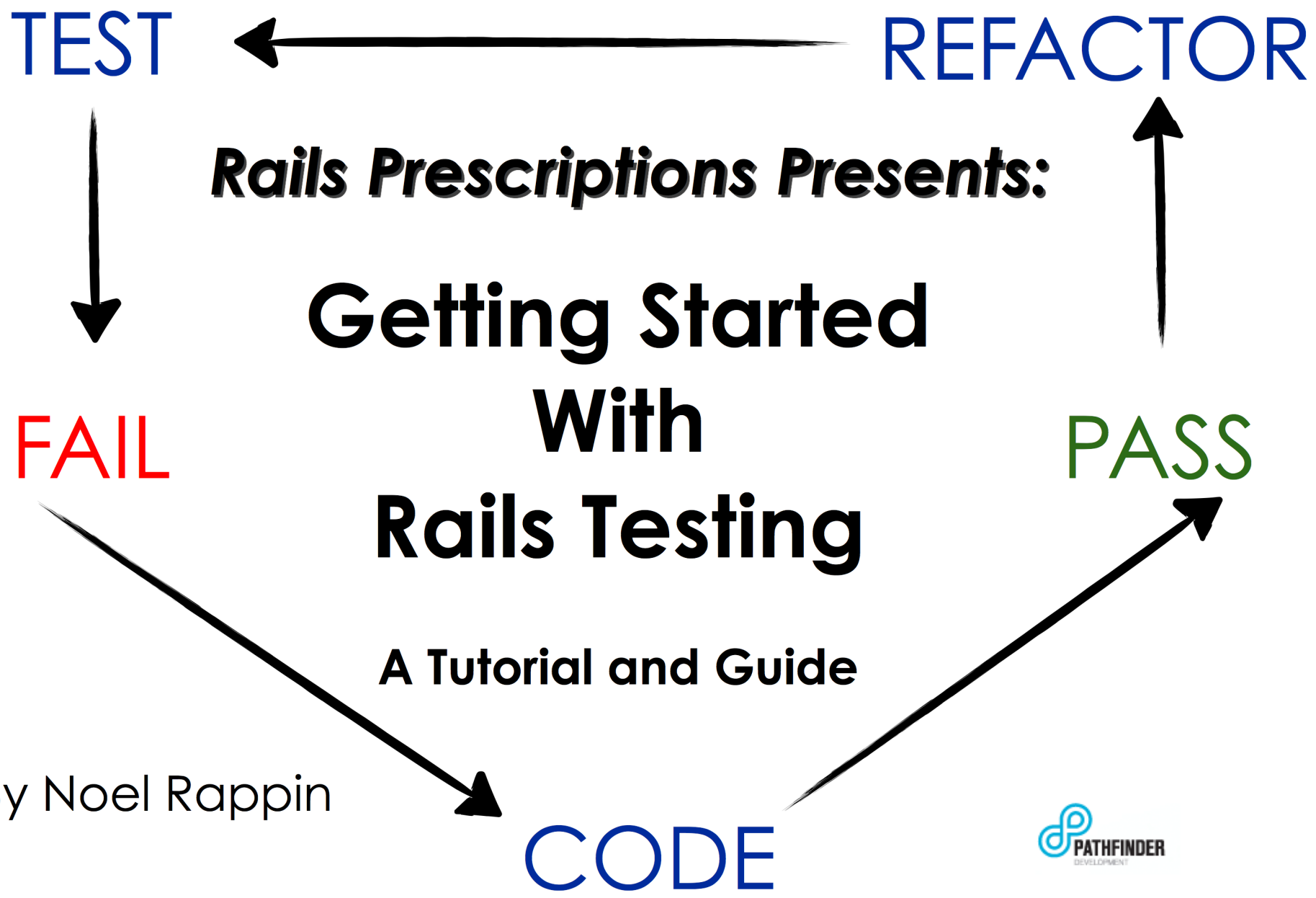
TEST
REFACTOR
Rails Prescriptions Presents:
Getting Started
With
Rails Testing
FAIL
PASS
A Tutorial and Guide
By Noel Rappin
CODE
PATHFINDER
DEVELOPMENT

Test Driven Development: A Tutorial

Version 1 January, 2009

The full version of this book will be released soon, see railsprescriptions.com for more details

Send comments to railsprescriptions@gmail.com

Chapter 1

Welcome To Rails Test Prescriptions

Thank you for your purchase of Rails Test Prescriptions. Here are a few things that you should know about this book.

What you have purchased

This book has no digital rights management or copy protection. As far as I am concerned, you have much the same rights with this file as you would with an actual book. I would appreciate if you would support this book by keeping the file off of public file-sharing servers. If you do wish to distribute this book throughout your organization, it would be great if you would purchase additional copies or support the book directly via the Donate button at http://www.railsprescriptions.com. Thanks.

Getting updates

This book is incomplete. New sections will be added, old sections will be updated, even the order of items may change. Your purchase entitles you to all updates of this book. However, you need to register your purchase at Rails Prescriptions to access updates. I'm sorry, this is more of a pain than I'd like for it to be.

To register your book purchase

- Visit the following url: http://www.railsprescriptions.com/users/new
- Enter your email and desired password. For support materials, please enter enough information from either a Lulu purchase or a PayPal donation to the site of $9 or more.
- You will receive an email shortly accepting your credentials. Updates are available at http://www.railsprescriptions.com/products

To be notified when an update is published, your best bet is to follow the blog at http://blog.railsprescriptions.com or follow the Twitter feed for the book at http://www.twitter.com/railsrx. You can also follow my personal Twitter feed at http://www.twitter.com/noelrap.

Finding mistakes and contacting the author

This book has mistakes. I just don't know what they are yet. If you have any feedback on this book, good, bad, or indifferent, you can contact the author via email at railsprescriptions@gmail.com. You can also submit support tickets to http://getsatisfaction.com/railsprescriptions. Mistakes will be corrected in the next published version.

Further information about the book is generally posted at the blog and twitter feed.

Getting Started With Testing Your Rails Application

One obstacle to becoming a test-driven Rails Ninja Warrior is the relative lack of decent tutorials about how and why to adopt test-driven practices in Rails. The "Create A Blog in 15 Minutes" kind of tutorials generally are focused on features rather than on testing, and the test-specific tutorials are pretty abstract and don't focus on real testing. That's a shame, because Rails has tremendous support for test-driven development, and using a test-driven process will dramatically improve the quality of your code and the speed with which you can develop new code.

In this guide, I'll be developing the first few features of a brand-new Rails application strictly test-first and explaining it all as I go.

Let's Build An Application

At my company, each project has a daily scrum where everybody on the project briefly describes what they did yesterday, what they plan on doing the following day, and if anything is blocking them from getting their work done.

Where is the boring stuff?

Take a look at the appendix at the back for the gory details of creating the application, getting it on Edge Rails, and adding the only plugin I'm going to use. Alternately, just check out the example from GitHub at http://github.com/noelrappin/huddle/tree/master .

I've tried to make it possible for you to follow this example in the source step by step, by using different branches in the git source repository. If you clone the source code for this example from GitHub (all one line):

```
git clone
  git://github.com/noelrappin/huddle.git
```

You'll find that each stopping point in the code corresponds to a git branch, so you can recreate the code at any point in this example with a command such as:

```
git checkout -b step_0 origin/step_0
```

This command creates a local branch, and associates it with the corresponding branch on the GitHub server. The branch `step_0` is the initial application, and `step_1` adds the initial data models.

Often, a team is geographically distributed, and we do these scrums via email. That's not impossible, but I think we can all do better with a little web application magic. The application will be called Huddle, and will support entering and viewing.

Some ground rules:

1. I'm assuming a basic understanding of non-test Rails concepts. You don't need me to tell you what a controller is. Similarly, I won't be talking much about things you can look up — the focus here is on process and rationale.

2. I'm working on Edge Rails, circa pre Rails 2.3. Most of this will be applicable in Rails 2.2, before that some of the syntax won't work.

3. I'm going to do this in a (mostly) strict test-driven style, meaning no code will be written in a controller, helper, or model except as part of Rails scaffolding or in response to a failing test. I'll be a little more lenient with view code.

4. In the interest of keeping this basic and helpful to the greatest number of people, I'm going to stick to core Rails features without any third-party test plugins or gems. RSpec, Shoulda, Machinist, and the rest will get some attention in the full book.

5. See what I did there? You'll probably notice references to topics that will described in more detail in the actual book. Bear with me. I want this sample to be useful, but I'd also like to sell some actual books.

6. Comments, complaints, deep thoughts, errata, and the like should be sent to mailto:railsprescriptions@gmail.com. If you want to be notified about news related to the release of the final book, and other stuff besides, check out the blog at http://blog.railsprescriptions.com and on Twitter as @railsrx or noelrap. Have fun.

The First Test First

This is the first thing I think I know about Huddle. It goes in `test/functional/status_reports_controller_test.rb` — if you've followed the appendix or started in git branch `step_1`, the file should already be there, just add this right in.

```
1  test "creation of status report with data" do
2    assert_difference('StatusReport.count', 1) do
3      post :create, :status_report => {
4        :project_id => projects(:one).id,
5        :user_id => users(:quentin).id,
6        :yesterday => "I did stuff",
7        :today => "I'll do stuff"}
8    end
9    actual = assigns(:status_report)
10   assert_equal(projects(:one).id, actual.project.id)
11   assert_equal(users(:quentin).id, actual.user.id)
12   assert_equal(Date.today.to_s(:db), actual.status_date.to_s(:db))
13   assert_redirected_to status_report_path(actual)
14 end
```

This test does the following:

- Line 3 simulates a `post` call to `status_report/create` with some various and sundry arguments.
- The block that starts in line 2 asserts that the number of `StatusReport` objects in the database is one more at the end of the call than it was at the beginning, using the `assert_difference` method.
- Line 9 uses the Rails `assigns` method, which allows access to instance variables set in the test controller, in this case, the controller variable `@status_report`. (You don't need the `@` as the argument to the method.)
- Lines 10, 11, and 12 assert that a project, user and status_date are added to the new object that is created.
- Line 13 asserts that the result of the controller call is a redirect to the show page of the newly created `StatusReport`.

Why Test? Reason 1

Writing code in conjunction with automated tests increases the quality of your code. Code written for tests contains small, loosely coupled pieces with few side effects. Amazingly, those features also make for code that is easy to read and maintain over time.

I do have some requirements here, but I don't want to dwell on them too much.

- A user is part of a project.
- A user can enter their scrum status every day, let's say, exactly once every day.
- Members of the project can see a timeline of status reports.

This test is about at the edge of what I would consider the complexity of a useful test, especially to start. In some cases, the amount of data means that you need some more complexity in setup or assertions. In practice, I'd combine the assertions a little bit with some helper methods that I've written over time, but for now, we're sticking to Rails core as much as possible.

The Classic Process

The way to succeed with test-driven development is to trust the process. The classic process goes like this:

1. Write a test
2. Write the simplest code that could possibly work
3. Refactor

Repeat until done.

I've chosen this as the starting place somewhat arbitrarily. It's a completely new feature in the application, and the first thing I know about it is that submitting the form will have a couple of consequences that I can test. One nice thing about test-driven development is that exact starting points don't matter that much. I could have started in the model and worked toward the controller. Right now, I'm more confident that I know what the controller should be doing, so I'm starting there.

I'm testing `status_date` because I know that there will need to be code to add that attribute to the object, and I'm testing the existence of the project and user objects because those require the relationships to be set up. I'm not testing the `today` and `yesterday` texts because that's part of core ActiveRecord — I could test it, but it would be redundant. That's not always bad, but right now it's unnecessary.

Some Rails testers would limit each test to a single assertion, and would therefore write this as four different tests. The advantage there is that each test would be able to pass or fail separately — as written, the first failure will prevent the rest of the tests from running. While sympathetic to the abstract point that assertions should be independent, I think it's easier to follow the intent of the test when similar assertions are grouped.

When I run the test via `rake`, I get a failure, the relevant part of the output 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:57:in `to_s'
```

Making The Process Work

This process tends to fail if you make it too complicated. The key is rapid feedback. Each step is small. Each new test is simple. The new code is simple. Most of your design or creative work comes in the refactor step. The order of the steps is less important than keeping the test and code continually in sync.

This is a slightly tricky error message — what's happening is that `actual.status_date` is `nil`, so `nil.to_s` would normally take no arguments. However, we expect the value to be a date, and `date.to_s` takes an optional format argument (at least it does in Rails with ActiveSupport).

Also, notice that the `user` and `project` parts of the test already pass. This is a Rails 2.2 and up thing — when I specified `user:references` in the `script/generate` command line, Rails automatically added the `belongs_to` to the `StatusReport` class. As we'll see later on, it didn't add the relationship in the other direction.

Now, let's make the test pass. The process says do the simplest thing that could possibly work. I like to make the immediate error or failure go away, even if I suspect there are further errors waiting in the test. Doing so helps avoid the worst kind of test problem, which is a test that mistakenly passes when it should really fail.

The beginning of the `create` method in `app/controllers/status_reports_controller.rb` is now changed to:

```
def create
  @status_report = StatusReport.new(params[:status_report])
  @status_report.status_date = Date.today          # ==> the new line
```

My First Refactor

This fixes the immediate problem, and the test passes. We now enter the refactoring step. There isn't much here, but I don't like setting the `status_date` in the controller. It should be set in the model.

Mantras

Don't try to solve all the problems at once. Don't look ahead to the next test when trying to make the current one pass. If you need more functionality, expose the problem in a test. If you get a bug report, expose the issue in a test. Don't refactor until all the tests pass.

That's the ideal, anyway. Reality will intrude from time to time.

Ordinarily, you would not be writing tests during refactoring, just using existing tests to verify that behavior hasn't changed. However, when moving code from one class to another, I do like to create tests in the new class. Especially here, because my new behavior will be slightly different — I 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, of course, fails. A quick note on line 4. Where possible, I always compare literal objects rather than composite ones, so both dates are converted to strings before comparison. Sometimes this will keep you from a test that is actually a tautology.

To pass the test, I'll to add a `before_save` callback to the `StatusReport` class:

```
1  class StatusReport < ActiveRecord::Base
2    belongs_to :project
3    belongs_to :user
4
5    before_save :set_status_date
6
7    def set_status_date
8      self.status_date = Date.today
9    end
10 end
```

One more thing, though — if some other object sets the `status_date` before save, that date should be used. Here's how to expose that requirement as a test in `test/unit/status_report_test.rb`:

```
1 test "saving a status report that already 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
```

A couple of notes about this code — the `to_date` methods are there to convert between `10.days.ago`, which is a `DateTime` object and the `status_date`, which is a `Date` object. Without that, you'll get an error because the string formats won't match in line 5.

The `reload` call forces ActiveRecord to retrieve the record from the database — in this particluar case, 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 you're testing and save — this is most commonly an issue in controller tests, where the controller might update the database for an object you've set up and are holding on to in the test.

The passing code is a very slight change to the model

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

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

One sec, while I take care of that… and it works, which you can verify by looking at the git branch `step_2`.

Avoid Pain

The most common way that testing starts to hurt is if you write too much code before trying to test it.

More Validations

While I'm looking at the model, there are a couple of other requirements for a valid status object that I'd like to cover. While it's possible for a user to

It seems counterintuitive and weird, but the more code is written up front, the harder it is to test. Integrating testing affects the design of the code in ways that makes further testing easier.

leave either half of the yesterday/today status blank, leaving both of them blank is not really a valid status. Back in `test/unit/status_report_test.rb`:

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

The simplest way to pass this test is as follows:

```
1 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:8]:
<"2008-12-26"> expected but was
<"">.

1) Failure:
test_should_create_status_report(StatusReportsControllerTest)
    [/test/functional/status_reports_controller_test.rb:16]:
<3> expected but was
<2>.
```

What? Well, you've probably figured it out, but adding the validation causes problems in other tests, namely that blank records that were being created by other tests that 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 other tests is straightforward. In `test/units/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
```

The test "saving a status report that already has a date doesn't override" needs a similar change. As does the test in `test/functional/status_report_controller_test.rb`:

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

And now we're back at all passing. This is, frankly, the kind of thing that causes people not to like testing, because 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 some things you can do to minimize the annoyance and keep on track with the benefits of working test first.

Why Test? Reason 2

The obvious reason: testing lets you verify the correctness of some part of your program. Automated testing lets you verify it without even trying. If your code was 100% perfect, and had 100% perfect tests, a user would never experience an error, and a programmer could not introduce a bug into your program without failing a test. In the real world, of course, total perfection is perhaps unattainable, but even the effort to get close pays real benefits.

The first is to keep a very tight loop between writing tests and writing code and run the test suite frequently (ideally, you'd run it constantly using autotest or a similar continuous test execution tool). The tighter the loop, the fewer lines of code you write in each back and forth, the easier it is to find and track down these structural test problems.

Secondly, and more specific to these kinds of validation problems, using some kind of factory or common 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 the larger book.

Anyway, that's a distraction, we have a larger problem. Remember, we wanted the status to only be invalid if *both* today and yesterday were blank. It's not a bad idea to write a couple of follow-up tests to confirm that we haven't overshot the mark, in `test/unit/status_report_test.rb`

```
1 test "a test with yesterday blank is valid" do
2   actual = StatusReport.new(:today => "today", :yesterday => "")
3   assert actual.valid?
4 end
5
6 test "a test with today blank is valid" do
7   actual = StatusReport.new(:today => "", :yesterday => "yesterday")
8   assert actual.valid?
9 end
```

Oops:

```
1   1) Failure:
2 test_a_test_with_today_blank_is_valid(StatusReportTest)
3     [/test/unit/status_report_test.rb:29]:
4 <false> is not true.
5
6   2) Failure:
7 test_a_test_with_yesterday_blank_is_valid(StatusReportTest)
8     [/test/unit/status_report_test.rb:24]:
9 <false> is not true.
```

I think at this point, I need to move to a custom validation, because the validation functions provided by Rails aren't quite going to get this right for me.

Replace the validation line in `app/model/status_report.rb` with the following:

```
validate :validate_has_at_least_one_status

def validate_has_at_least_one_status
  if today.blank? and yesterday.blank?
    errors.add_to_base("Must have at least one status set")
  end
end
```

And we're back to passing. This point corresponds to git branch `step_3`.[1]

A couple of points on the question of what to test and when.

- While I don't need to test the Rails validation methods as such, I 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 that causes me 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 a little bit of an open question. As a matter of course, I insert a generic test into my controller scaffold using mock objects to cover the failure case (more in the book), which means I don't feel the need to go back and cover every different kind of model failure. For sure, you don't need to test the

1. One early reviewer wanted me to point out this can be done as a pair of `validates_presense_of` calls with the `if` option, instead of with a custom method. That's fine, too.

controller behavior for each and every different possible model failure. Unless, of course, each specific failure actually dictates different controller behavior.

Now, this may seem like a lot of work because I'm a wordy pain in the neck, and I've been describing 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.

Security, Now

Before we go too far down the functionality path, I'd like to introduce the login and security models via RESTful Authentication. RESTful authentication has its own set of tests, so we don't need to write for the basic behavior of login and log out. We do need to write tests to cover parts of the application-specific security model for who can see and edit what different things. Here's what I've got:

Why Test? Reason 3

A complete automated test suite will prevent regression bugs later on. If you or somebody else accidentally makes a code change that breaks existing code, it will be caught by existing tests.

1. A user must be logged in to view or create a status report.
2. A user will 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. A user can only edit their own reports. Again, there may or may not be admin functionality later, and we'll cross that bridge when we get to it.

In order to use any RESTful Authentication features, you need to add the following inside the `ApplicationController` class. In a slight break from normal procedure, we'll implement the forced login in the code first.

```
include AuthenticatedSystem
before_filter :login_required
```

The sessions controller needs the following line to remove this global filter, otherwise you'll be requiring a user to log in in order to be able to log in. Which makes for a neat Escher painting, and an annoying application.

```
skip_before_filter :login_required
```

Why not do this test first? Because most of the functionality is already tested by RESTful Authentication, and because the 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) then I'd start adding some tests.

For another reason, 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:

```
29 tests, 26 assertions, 17 failures, 6 errors
```

The test failures are all due to the login requirement — every controller test is now being redirected off to the login page.

To fix this globally, put the following inside `ActiveSupport::TestCase` in `test/test_helper.rb`, this gives you access to methods for faking a user login:

```
1 include AuthenticatedTestHelper
2
3 def login_quentin
4   login_as :quentin
5 end
```

After that, the three controller tests where the controller requires a login (`projects_controller_test`, `status_reports_controller_test`, and `users_controller_test`), all need the following line inside the test class at the top:

```
1 setup :login_quentin
```

This is the first time we've used the setup/teardown mechanism as implemented in Rails 2.2 and up. Plus, we're using RESTful Authentication's scheme for faking logged in users in tests.

Here's what's happening — the `login_quentin` method uses the `login_as` method provided by RESTful Authentication to fake a user by setting the test session object directly. Within each controller, the `setup :login_quentin` line makes that method a setup method in exactly the same way that a Rails controller can set up a `before_filter` — all methods designated as `setup` are called before each and every test. All methods designated as `teardown` are called after each and every test. Setup and teardowns are invoked in the order declared — which is why you can't just put this `setup` line in the test helper, if you try to, then the `login_quentin` method is invoked before the setup in the actual controller class, meaning the test session object is not yet available.

At this point the tests pass, and we're at git branch `step_4`.

Applying Security

Now that user login is required, the form for creating a status report should no longer have the `user_id` as an entry in the form, the currently logged in user is assumed to be the creator. Similarly, although we haven't specified a mechanism for setting it, the `project_id` will always be an implicit current project, and also doesn't need to be specified in a form. Which leads us to a new test… well, actually an edit of an existing test. In `test_helper.rb`, add the following helper method.

```
1 def set_current_project(symbol)
2   @request.session[:project_id] = projects(symbol).id
3 end
```

Then change the previously written status report test to remove the project and user from the test we wrote a few pages ago:

```
 1 test "creation of status report with data" do
 2   set_current_project(:one)
 3   assert_difference('StatusReport.count', 1) do
 4     post :create, :status_report => {
 5       :yesterday => "I did stuff",
 6       :today => "I'll do stuff"}
 7   end
 8   actual = assigns(:status_report)
 9   assert_equal(projects(:one).id, actual.project.id)
10   assert_equal(users(:quentin).id, actual.user.id)
11   assert_equal(Date.today.to_s(:db), actual.status_date.to_s(:db))
12   assert_redirected_to status_report_path(actual)
13 end
```

The test helper method created in the first snippet, then called in line 2 above sets the current project in the test session — for the moment we don't care that there's no UI way to set that project.

Now, the first time I tried to write this section I actually wrote this as a separate test — as a general rule, I don't like to edit existing tests unless they are actually broken by some change to the code. In this case, though, the behavior of the original test is really dangerous — we don't want the `user_id` from the form to be acknowledged at all, to prevent a malicious user from posting nasty things under your good name. So I chose to edit the existing test.

Preventing the malicious user is the next test, let's pass this one first. The `create` method in `StatusReportsController` needs to be changed to start:

```
def create
  params[:status_report].merge!(:user_id => current_user.id,
      :project_id => current_project.id)
  @status_report = StatusReport.new(params[:status_report])
```

The `current_user` method is defined by RESTful Authentication, and since we are requiring a login to get this far is guaranteed to be non-nil. We also need a `current_project` that will always be non-nil. Put this in the `ApplicationController`:

```
private
helper_method :current_project
def current_project
  project = Project.find(session[:project_id]) rescue Project.last
end
```

If there is no `project_id` in the session, then the method returns the most recently created project. That's almost certainly not the final logic, but we can get by with it for now.

The tests pass and this is git branch `step_5`.

Punishing Miscreants

Now the time has come to punish evildoers who try to get around the site by putting somebody else's `user_id` into their form submit. Right now, based on the above code, the submitted user name is ignored and the actually logged in user is used instead. I'd rather actually kick out a user trying to fake a `user_id`. The test, in `status_reports_controller_test.rb` looks like this:

```
 1 test "redirect and logout if the user tries to snipe a user id" do
 2   set_current_project(:one)
 3   assert_no_difference('StatusReport.count') do
 4     post :create, :status_report => {
 5       :user_id => users(:aaron).id,
 6       :yesterday => "I did stuff",
 7       :today => "I'll do stuff"}
 8   end
 9   assert_nil session[:user_id]
10   assert_redirected_to(login_path)
11 end
```

There are a few differences between this test and the previous one. In line 5, I've added a `user_id` to the params for the user named Aaron (remember that the logged in user is Quentin). In line 3, I'm explicitly testing that `StatusReport.count` does not change — that no new report is created. Finally, lines 9 and 10 assert that the user is logged out and bounced back to the login page.

There are several things about that test that will fail at the moment (though I still think it makes sense long term to keep it as one test). I like to fix these things one at a time. The first failure is the `assert_no_difference` call, so we need to prevent the creation of the new object.

My first attempt is a) ugly and b) doesn't work:

```
1  def create
2    if params[:status_report][:user_id].nil? ||
3          (params[:status_report][:user_id] == current_user.id)
4      params[:status_report].merge!(:user_id => current_user.id,
5          :project_id => current_project.id)
6      @status_report = StatusReport.new(params[:status_report])
7    end
8    respond_to do |format|
9      if @status_report && @status_report.save
10       flash[:notice] = 'StatusReport was successfully created.'
11       format.html { redirect_to(@status_report) }
12       format.xml  { render :xml => @status_report, :status => :created,
13           :location => @status_report }
14     else
15       format.html { render :action => "new" }
16       format.xml  { render :xml => @status_report.errors,
17           :status => :unprocessable_entity }
18     end
19   end
20 end
```

The attempt here is in lines 2-6 to only create the object if there is a user id and if it differs from the current user, then make the existence of the object a prerequisite for the save test in line 9.

It doesn't work because in the failure case, the code tries to render the `new` action, which assumes the existence of `@status_report`. I could try to make that a redirect in line 14, but that's getting down a rathole that I'm only going to walk away from in just a second. So let's try a more general solution.

```
1  def create
2    if (params[:status_report][:user_id] &&
3      params[:status_report][:user_id] != current_user.id)
4      logout_killing_session!
5      redirect_to login_path
6      return
7    end
8    params[:status_report].merge!(:user_id => current_user.id,
9        :project_id => current_project.id)
10   @status_report = StatusReport.new(params[:status_report])
11   respond_to do |format|
12     if @status_report.save
13       flash[:notice] = 'StatusReport was successfully created.'
14       format.html { redirect_to(@status_report) }
15       format.xml  { render :xml => @status_report, :status => :created,
16           :location => @status_report }
17     else
18       format.html { render :action => "new" }
19       format.xml  { render :xml => @status_report.errors,
20           :status => :unprocessable_entity }
21     end
22   end
23 end
```

The tests now all pass. The `if` statement in lines 2 and 3 grab invalid forms, and the logout and redirect happen there, with the `return` on line 6 preventing further processing.

I'd like to make this a more general method — the problem is that if I just convert lines 2 – 7 into a helper method, the return statement still needs to be in the actual controller action or else the action will continue to process and you'll get a double-render error. The solution looks a bit like this.

The general method goes in the `ApplicationController`. It performs the same test against the current user's id, does the logout and redirect if they don't match, and returns `true`.

```
1 def redirect_if_not_current_user(user_id)
2   if user_id && user_id != current_user.id
3     logout_killing_session!
4     redirect_to login_path
5     return true
6   end
7   false
8 end
```

The controller method now needs to take that result and use it to stop it's own processing:

```
1  def create
2    redirect_if_not_current_user(params[:status_report][:user_id]) and return
3    params[:status_report].merge!(:user_id => current_user.id,
4        :project_id => current_project.id)
5    @status_report = StatusReport.new(params[:status_report])
6    respond_to do |format|
7      if @status_report.save
8        flash[:notice] = 'StatusReport was successfully created.'
9        format.html { redirect_to(@status_report) }
10       format.xml  { render :xml => @status_report, :status => :created,
11           :location => @status_report }
12     else
13       format.html { render :action => "new" }
14       format.xml  { render :xml => @status_report.errors,
15           :status => :unprocessable_entity }
16     end
17   end
18 end
```

Line 2 is the key here — it calls the general method then returns if the result is true. I find that version the most readable, even if it's a bit Perl-ish for my general taste, you could also try:

```
1 return if redirect_if_not_current_user(params[:status_report][:user_id])
```

Or,

```
if redirect_if_not_current_user(params[:status_report][:user_id])
  return
end
```

Or, using a `before_filter` at the top of the controller

```
before_filter :redirect_if_not_current_user, :only => [:create]
```

In which case, you don't need anything in the `create` method, but the helper method needs to flip and return `false` if the redirect happens in order to stop further processing and `true` if the user is OK.

And with that refactoring, we're at git branch `step_6`.

Testing for security is generally a very good use of time — security tests are relatively easy to write, since in many cases you're testing for just a redirection to a login page or that key data doesn't display. The trick is to be creative and try to think of all the oddball ways that somebody might be able to submit data to flummox your application.

Now, For A View Test

When testing a view, it's important to keep the your larger goals in mind because it's very easy to fritter away huge amounts of time trying to get the view tests ultra-complete and correct over time. It's easy to write view tests that are over-detailed and add little to the amount of coverage in your test suite, but will be prone to breaking any time the web designer looks at the code cross-eyed.

Why Test? Reason 4

Tests can provide very useful documentation of the expected behavior of your application. This is especially true for unusual or corner cases that would normally be ignored by documentation.

These issues will be covered more completely in the full book, but here are three tips to keep in mind as we embark on two sets of view tests.

1. Consider moving view logic to helpers where possible. It's much easier in the core Rails setup to individually test a helper than it is to individually test an view partial. On the other hand, moving view code from HTML/ERb and into Ruby methods could cause a web-only design team to come after you with torches and pitchforks, so beware.

2. Remember that the purpose of view testing is for the developer to validate the logical correctness of the application. Automated view testing isn't (and in Rails, really can't be) a substitute for walking though the site with your eyes open and checking that things line up and look nice.

3. Test at the semantic level, not the display level. Liberally assign DOM ids to the elements of your view and test for the existence of the DOM ids rather than the actual text. Obviously, there are limits here — sometimes the actual text is critical. But the advantage of DOM id testing is that it doesn't break every time the content team updates the text or when the designer changes the CSS.

There are two view tests that should be written to close out this feature. I'd like to validate that the form being generated actually contains the field names that the create method expects — this little gap is one of the easiest ways for a problem to slip through the cracks of an otherwise air-tight Rails TDD process. Then I'd like to test that the project show page actually displays the timeline of status reports for the project.

The form test just validates that the form elements actually exist in the form as I want them too, which means that the elements that will be automatically set by the controller shouldn't be in the form. With

core Rails test structures, this test goes with the controller in `test/functional/status_reports_controller_test.rb`

```
1  test "new form has expected elements" do
2    get :new
3    assert_select "form[id=new_status_report][action=/status_reports]" do
4      assert_select "#status_report_project", :count => 0
5      assert_select "#status_report_user", :count => 0
6      assert_select "textarea#status_report_today"
7      assert_select "textarea#status_report_yesterday"
8      assert_select "#status_report_status_date_1i", :count => 0
9    end
10 end
```

This is a reasonably straightforward view test which simulates a `GET` request for the new object form in line 2 and then uses `assert_select` to verify several features of the output of the call. `assert_select` has about seventy-eleven different options, all of which are lovingly detailed in the full book. For now, the basic point is that the first argument to `assert_select` is a CSS-like selector, and the remaining arguments are assertions about elements in the output that match the selector.[2] For our purposes, the most useful options are `:text`, which is either a string or regular expression. If specified, the internal contents of tags that matches the selector are checked to see if any matches the argument. The other option is `:count`, which is the number of elements that will match the selector. If `:count` is not specified, then the expectation is that there will be at least one match. If `:count` and `:text` are specified then `:count` is the number of items that both match the selector and the text.

2. One handy feature of `assert_select` is that it parses the HTML markup, and will spit out a warning if the HTML is badly formed, such as by missing an end tag.

Also, `assert_select` calls can be nested using blocks, which adds the additional constraint that the assertions inside the block must all be true inside the body of a tag that matches the outer `assert_select`. (This can also be achieved by using compound selectors like `ul > li`.) In this case, the outer `assert_select` matches a `form` tag with the id and action that you would expect by Rails convention for a new action. All the other `assert_select` calls match elements that must also be inside that `form` tag.

Why Test? Reason 5

Testing your code gives you a place to start. It's extremely valuable on a complex or vaguely defined feature to be able to just put one flag in the ground and say "I know this is true". Generally, the following steps become clearer once the first one is taken.

Specifically, I'm testing that there will not be a project, user, or date tag of any kind, and I'm also testing that there will be text area entry tags for the yesterday and today status report fields.

You could quibble back and forth on the exact structure of these tests. I've chosen to test the form fields based on their DOM ids, knowing that Rails has a consistent pattern for ids and the name field of the tag. DOM ids work a little bit better with the `assert_select` syntax.[3]

Still, a clueless programmer who was breaking the Rails conventions could cause a bug without failing this view test. That said, chasing down every way the view could break would take all your effort forever — I try to limit my view testing to things that are likely sources of error or regression.

Passing the test is quite easy, just remove the form elements for the project, user, and date. However, unlike the model tests, even with the passing test, this view is in no way, shape, or form fit to be in a production application. You need styling, the Rails scaffold HTML is not optimal, that sort of thing. In the interest of keeping this tutorial under 100 pages, I'm not going to go into the details.

And that's the end of git branch `step_6`.

3. At least two early reviewers think I'm crazy here, and these tests should track the name directly, as in `assert_select "textarea[name = ?]", "status_report[today]"`.

At this point in a "real" development environment, the feature is essentially finished, but you *must* go into the browser and test it, no matter how complete you think your tests are. Tests are great and wonderful, but the user isn't going to care about your code coverage if it doesn't work in the browser. Writing tests minimizes the amount of time you'll spend cycling through the browser, but in no way does automated testing mean you can stop using the browser.

As for us in tutorial world, let's move on to the display functionality for an entire project.

Testing the Project View: A Cascade of Tests

The project page should show a timeline of status reports for that project. This functionality will require at least a controller and a view test to start, and we may use some model tests later on. We are going to need some status report data for this test, and we haven't discussed fixtures yet. Generically, a **fixture** is any pre-defined set of data that is used by multiple tests as a baseline. Specifically in Rails, every ActiveRecord model gets an associated set of fixture data in `test/fixtures`, where data objects can be specified in YAML format.

Rails fixtures are in a nested format, at the top level each individual model gets a name followed by a colon, the data is indented Python-style, and the keys must be columns in the database table for that model (meaning that they can't be arbitrary methods in the Ruby code they way they can be when calling `new` or `create`). Add this to the `test/fixtures/projects.yml` file to create a single project.

```
1 huddle:
2   name: Huddle
3
```

This object is loaded into the database before every test,[4] and is by default accessible anywhere in any test using the method call `projects(:huddle)`.

We'll also need some status reports. Here are two for user Quentin, these go in `test/fixtures/status_reports.yml`

```
quentin_tue:
  project: huddle
  user: quentin
  yesterday: Worked on Huddle UI
  today: Doing some testing
  status_date: 2009-01-06

quentin_wed:
  project: huddle
  user: quentin
  yesterday: Did Some Testing
  today: More Testing
  status_date: 2009-01-07

```

There are a couple of things to note about the fixtures. First is that when I said that all the keys had to be database columns, that was a partial lie, they can also be associations. If the key is an association, the value represents the name of a fixture in the related table (or a comma-separated list of features for a

4. Well... it's a little more complicated than that, see the full book.

one-to-many relationship). You also do not need to specify an `id`, Rails will automatically generate one for you. (If you do specify an `id`, the nifty auto association feature will not work).[5].

This means, by the way, that you need to go into the `test/fixtures/users.yml` file and delete the `id` line in each and every user fixture in that file. RESTful Authentication uses the older style of specifying ids explicitly, and if you don't make this change, the fixture. connections will not work correctly

As you can also see, dates are automatically converted from string representations. There are a couple of other tricks to using fixtures, and some places where having one set of global data totally breaks down. See the full book for more detail.

I'll also add two more in the same file for user Aaron:

By the way, if you are copy and pasting these into a text file, remember that YAML files require a specific whitespace layout — the outdented fixture names need to be in the leftmost column of the line.

5. All this fancy `id` mapping is a Rails 2.0 and up feature. Before then, you had to track the `id` values manually. Which was a total pain for many-to-many join tables.

```
aaron_tue:
  project: huddle
  user: aaron
  yesterday: set up huddle schema
  today: pair programming with quentin
  status_date: 2009-01-06

aaron_wed:
  project: huddle
  user: aaron
  yesterday: sick
  today: trying to pair again
  status_date: 2009-01-07

```

Give your fixtures meaningful names, it'll help later on — the Rails default names are a pain to keep straight.

With fixture data in place, we can start testing. The controller method for project show should gather the reports for that project and group them by date. This goes in `test/functional/projects_controller_test.rb`

```
1 test "project timeline index should be sorted correctly" do
2   set_current_project(:huddle)
3   get :show, :id => projects(:huddle).id
4   expected_keys = assigns(:reports).keys.sort.map{ |d| d.to_s(:db) }
5   assert_equal(["2009-01-06", "2009-01-07"], expected_keys)
6   assert_equal(
7       [status_reports(:aaron_tue).id, status_reports(:quentin_tue).id],
8       assigns(:reports)[Date.parse("2009-01-06")].map(&:id))
9 end
```

This asserts in lines 4-5 that an object called `@reports` is created, and that its keys are the dates of the reports that are found. Lines 6-8 assert that each key contains its reports, sorted by the name of the user (well, technically, the email of the user… the user model created by RESTful Authentication doesn't have name fields, and we haven't added them yet, because again, I'm trying to keep this under a zillion pages).

That test will, of course, fail, because `assigns(:reports)` is `nil`.

Moving to the controller itself, I pretty quickly decide to just defer the assignment to the model. In `app/controllers/project_controller.rb`

```
def show
  @project = Project.find(params[:id])
  @reports = @project.reports_grouped_by_day
  respond_to do |format|
    format.html # show.html.erb
    format.xml  { render :xml => @project }
  end
end
```

Which means I now need a model test in `test/unit/project_test.rb`:

```
test "should be able to retrieve projects based on day" do
  actual = projects(:huddle).reports_grouped_by_day
  expected_keys = actual.keys.sort.map{ |d| d.to_s(:db) }
  assert_equal(["2009-01-06", "2009-01-07"], expected_keys)
  assert_equal([status_reports(:aaron_tue).id, status_reports(:quentin_tue).id],
      actual[Date.parse("2009-01-06")].map(&:id))
end
```

As you've probably noticed, this is a direct swipe of the code I just put in the controller test. This opens up the question of whether I need both the controller and model test here. The model test is important because it's closest to the actual implementation and is the easiest to write error case tests on. The controller test adds just the piece of information that the model method is, in fact, called by the controller. This is a classic place for a mock object call in the controller, which removes the duplicate test on the return value of the model method. Mock objects are a huge topic in their own right and are covered in more detail in the full book. For our purposes here, the duplicate test will be fine.

Anyway, passing this requires a couple of things. In `app/models/project.rb`:

```
1 class Project < ActiveRecord::Base
2
3   has_many :status_reports
4
5   def reports_grouped_by_day
6     status_reports.by_user_name.group_by(&:status_date)
7   end
8
9 end
```

Previously, this model had been blank — I added the `has_many` line here because it's needed to pass the test. Note that I don't need to separately test what `has_many` does — that's part of the Rails framework itself. The need for the association line came immediately as I started writing tests for functionality of that model.

The actual method uses the Rails ActiveSupport `group_by` which returns the hash structure we want. The `by_user_name` is a named scope inside `StatusReports` that sorts the reports based on the user email. I actually wrote the shell of this method, then realized the named scope was going to be complex enough to require its own test. In `test/unit/status_report_test.rb`:

```
1 test "by user name should sort as expected" do
2   reports = StatusReport.by_user_name
3   expected = reports.map { |r| r.user.email }
4   assert_equal ["aaron@example.com", "aaron@example.com",
5         "quentin@example.com", "quentin@example.com"], expected
6 end
```

I should mention that I only did this because the named scope crosses multiple tables. For a simple scope, I probably wouldn't have bothered, since the functionality is mostly covered by Rails and is also easily tested by the existing test.

There are two crossing issues here: when to test functionality provided by the framework, and when you need to test private or subordinate methods that are only called by other tested methods. For instance, if I refactor a smaller method out of a larger, already tested method, I rarely feel the need to also write targeted tests against the smaller method — if the smaller method is simple, then it's functionality is adequately tested by the test on the larger method. Again, the more complex the subordinate method is, the more likely I am to write a separate test. For instance, if I find a bug in the smaller method, then it gets tests immediately.

The scope in `app/model/status_report.rb` looks like this:

```
1 named_scope :by_user_name, :include => "user",
2     :order => "users.email ASC",
3     :conditions => "user_id IS NOT NULL"
```

And now all the tests pass.

So this actually is the order I wrote this code in — controller test, model test in `Project`, Project model implementation (after checking that `group_by` would do what I want, I wrote the one liner directly), then scope test, then scope code (with a little flailing in there about exactly how to manage the users table). I don't know whether that's strictly test first, but I do know that in the entire process I never wrote more than about five lines of code on one side of the test/code divide without jumping to the other, and I never went more than a minute or two without running the test suite.

And that's the key to success with TDD — keep a tight feedback loop between the code and tests, don't ever let one or the other get too far out in front.

Now, we need a genuine view test to validate that something reasonable is going into the view layer, back in `test/functional/projects_controller_test.rb`.

```
test "index should display project timeline" do
  set_current_project(:huddle)
  get :show, :id => projects(:huddle).id
  assert_select "div[id *= day]", :count => 2
  assert_select "div#2009-01-06_day" do
    assert_select "div[id *= report]", :count => 2
    assert_select "div#?", dom_id(status_reports(:quentin_tue))
    assert_select "div#?", dom_id(status_reports(:aaron_tue))
  end
  assert_select "div#2009-01-07_day" do
    assert_select "div[id *= report]", :count => 2
    assert_select "div#?", dom_id(status_reports(:quentin_wed))
    assert_select "div#?", dom_id(status_reports(:aaron_wed))
  end
end
```

Continuing with the idea of semantic-level tests for the view layer, this test checks to see that there is some kind of `div` tag for each day, and that inside that tag is a `div` tagged for each status report. Even putting in a `div` might be over-specific — it's possible these might be table rows or something. The `count` tests are to prevent against more content showing up than expected, which is not caught by just testing for the existence of known tags. In this case, I'm also testing that there are exactly two days worth of reports, and that each day has two reports. This is calibrated to match the fixture data I just set up, and gives a quick look at one weakness of fixtures — a change to that fixture data could break this test.

Also, for this to work, add the line

```
1 include ActionView::Helpers::RecordIdentificationHelper
```

inside the class definition in `test/test_helper.rb` — this allows you to use the `dom_id` method in tests, which is handy.

Allowing that there are a jillion ways that the view code could pass the letter of this test and violate the spirit, here's the basic structure of a passing view, in `app/views/projects/show.html.erb`.

```
 1 <h2>Status Reports for <%= @project.name %></h2>
 2 <% @reports.keys.sort.each do |date| %>
 3   <div id="<%= date.to_s(:db) %>_day">
 4     <h3>Reports for <%= date.to_s(:long) %></h3>
 5     <% @reports[date].each do |report| %>
 6       <div id="<%= dom_id(report) %>">
 7         Yesterday I: <%= h report.yesterday %>
 8         Today I will: <%= h report.today %>
 9       </div>
10     <% end %>
11   </div>
12 <% end %>
```

This gives the view a `div` for each date, and a nested `div` for each individual report.

The view test passes and takes us to the end of git branch `step_7`.

The End of The Beginning.

That's the end of this getting started in testing guide. I've only scratched the surface of what's possible and valuable in Rails testing, and I hope you check out the full book *Rails Test Prescriptions: Keeping your application healthy* when it becomes available. The full book contains sections on all kinds of topics related to Rails testing. Sections include:

- Coverage of all the different kinds of tests provided by Rails.
- How to write good, well-formed, and useful tests
- How to test other parts of your application, such as views, file uploads, Ajax calls, and mailers.
- How to deal with a legacy codebase without tests.
- Using third party test plugins for different test structures, or to replace fixtures, or to add easier testing syntax
- Using mock objects and other test doubles.
- Using coverage tools to measure the reach of your tests.

In the meantime, I'm committed to keeping this guide up-to-date, error-free, and useful. If you have concerns about anything here, please send an email to railsprescriptions@gmail.com or leave feedback on the web site at http://www.railsprescriptions.com.

Also, if you liked it, telling all your friends would be a great help. If you didn't like it, tell them anyway.

If you'd like to follow the news on upcoming books, plus other issues related to Rails, technical publishing, or whatever, please checkout the Rails Prescription 24-Hour Window Blog at http://blog.railsprescriptions.com.

Shorter and perhaps pithier updates can be found at http://www.twitter.com/noelrap.

Thanks!

Acceptance Testing With Cucumber

This one is going to be structured a little differently. More narrative and exploratory. As I start writing this, I'm not completely sure exactly how I feel about Cucumber. On the one hand, it's a potentially great way to do a lot of acceptance testing, and especially to enable clients to sign off on your acceptance tests. On the other hand, it's not hard to see this become a fantastic way to spin your wheels while feeling like you are accomplishing something. I think there's a sweet spot, so let's try and find it.

Getting Started

Cucumber is distributed as a gem. As I write this, the stable gem is 0.3.0 and is installable as:

```
gem install cucumber
```

Eventually, the developer's version will probably get ahead of the stable gem, you can download that version with the following:

```
gem install aslakhellesoy-cucumber
```

You could also include this in your `environment.rb`. I'm adding this to a branch of the Huddle project called `cucumber`, on the theory that I'd like the source repository to run with no outside dependencies. Then a `rake gems:install` and a `rake gems:unpack:dependencies` will put cucumber in the `vendor/gems` directory, along with a bunch of other dependencies that we're not going into right now. You can install this code by switching to the Git branch `cucumber` in the GitHub source for huddle at `http://github.com/noelrappin/huddle/tree/master`

In order to get the full effect out of Cucumber, you'll also want to install Webrat, which will be fully discussed in a later prescription. Webrat is a browser simulator that can be used to create acceptance tests on its own, but which is also called by Cucumber to run its own acceptance tests. The most up-to-date version of Webrat is currently distributed as a plugin.

```
script/plugin install git://github.com/brynary/webrat.git
```

Cucumber was designed to be part of RSpec, and although you don't need to be using RSpec for Cucumber to work, some of the default steps use RSpec-style matchers. Having the RSpec and RSpec-Rails gems installed in your development environment should be enough to get the examples in this section to run.

To start using Cucumber for Rails testing, you need to generate some files:

```
$ script/generate cucumber
  create  features/step_definitions
  create  features/step_definitions/webrat_steps.rb
  create  features/support
  create  features/support/env.rb
  create  features/support/paths.rb
  exists  lib/tasks
  create  lib/tasks/cucumber.rake
  create  script/cucumber
```

What you get here is a `features` directory where your actual cucumber features will go with two subdirectories. One of them contains some setup code, the other will take your cucumber step definitions, about which more in a moment. You also get a Rake file, and a `cucumber` script which lets you run one Cucumber feature at a time.

If you absolutely don't like using RSpec matchers, go into the newly created `features/support/env.rb` and comment out the following two lines:

```
#require 'cucumber/rails/rspec'
#require 'webrat/core/matchers'
```

However, some of the default Webrat steps use the RSpec matchers, so I recommend having the RSpec gems around for Cucumber even if you don't use RSpec.

And now we're good to go.

Writing features

What Cucumber allows you to do is write acceptance tests for new features in a lightly structured natural-language format, then convert those tests into executable Ruby code that can be evaluated for correctness. We'll talk more about how this might affect your workflow in a little bit, first let's go through an example.

Right now, the Huddle application doesn't have much of a user interface. There's no way to associate a user or a user's status reports with a project, for example. So, let's create one. From the huddle command line:

```
$ script/generate feature users_and_projects
  exists  features/step_definitions
  create  features/manage_users_and_projects.feature
  create  features/step_definitions/users_and_projects_steps.rb
```

The Cucumber generator gives you a feature file where you will put the actual Cucumber code Technically, Cucumber is the entire system. The feature language is called Gherkin. Which is an outstandingly fun word to say over and over again. and a ruby file where you will put the step definitions that bridge the gap between Cucumber and your actual Rails application. There's also some boilerplate code which is useful if you are trying to do acceptance test for basic CRUD functionality. Which we aren't, so I'm deleting it all. There's related boilerplate in the step definition file, that's going away as well.

Instead, I'm going to describe the feature I do want. A cucumber feature starts with a header:

```
Feature: Add users to project
  In order to make this program even minimally useable
  Pretty much everybody on the planet
  wants to be able to add users to a project
```

This is strictly for the humans — Cucumber ignores all the header parts when processing.

After the header, there's an optional background section.

```
  Background:
    Given a project named "Conquer The World"
    And the following users
      | login| email             | password| password_confirmation|
      | alpha| alpha@example.com| alpha1  | alpha1               |
      | beta | beta@example.com | beta12  | beta12               |
```

In case it's not clear, the `Background` line is indented in-line with the `In order to` lines, meaning that it's two spaces in from the left side. The `Background` statements are analogous to a startup block and are evaluated before each scenario being tested. I'll talk about the specific syntax in a sec.

An individual unit of a Cucumber feature is called a scenario. I've defined two. The first goes to the edit page display for a project.

```
Scenario: See user display on edit page
  Given that user "alpha" is a member of the project
  When I visit the edit page for the project
  Then I should see 2 checkboxes
  And the "alpha" checkbox should be checked
  And the "beta" checkbox should not be checked
```

And the second goes to what happens when the edit form is actually submitted.

```
Scenario: See users in project edit
  Given I am on the edit page for "Conquer The World"
  When I check "alpha"
  And I press "Update"
  Then I am taken to the show page for "Conquer The World"
  And I should see "alpha@example.com"
  And I should not see "beta@example.com"
```

There are several things to say about these cucumbers.

A Cucumber scenario has a basic structure with three parts — the givens, which indicate preconditions to the action; the when lines, indicating a user action that changes state; and the then lines, containing the result of the state change. A line beginning with `And` belongs to it's most immediate predecessor clause. Obviously, the items in the `Background` clause are all expected to be givens. As for the rest of each clause, that's pretty much free-form, it's the step definitions that give those structure. Oh — the thing that looks like a table in the `Background` clause. Guess what? It's a table, and that's the preferred method of defining a set of data for a Cucumber feature, rather than using fixtures.

As it happens, this feature can be executed right now. It's not going to do much, though. Here is enough of the output to get the gist.

Each scenario gets a step by step running, if your terminal accepts it, the lines are color coded for success, failure, skipped due to an earlier failure, and undefined.

```
$ rake features
Scenario: See user display on edit page
      # features/manage_users_and_projects.feature:13
    Given that user "alpha" is a member of the project
      # features/manage_users_and_projects.feature:14
    When I go to the edit page for the project
      # features/step_definitions/webrat_steps.rb:10
    Then the "alpha" checkbox should be checked
      # features/step_definitions/webrat_steps.rb:101
    And the "beta" checkbox should not be checked
      # features/manage_users_and_projects.feature:17
```

At the end, you get a summary:

```
2 scenarios
4 skipped steps
9 undefined steps

You can implement step definitions for missing steps with these snippets:

Given /^a project named "Conquer The World"$/ do
  pending
end
```

Writing Step Definitions

Step definitions need to be created for each undefined step. We can paste them directly into the `features/step_definitions/users_and_projects_steps.rb` file. At least as a start.

As you can see from that sample, each step definition starts with a regular expression that defines what steps match it. When a matching step is found, the block attached to that step is executed.

Groups in the regular expression are passed as arguments to the block, which enables a single step definition to match many different steps. So, the above step definition is probably more useful if it's written like this:

```
1 Given /^a project named "(.*)"$/ do |project_name|
2   @project = Project.create!(:name => project_name)
3 end
```

The regular expression group notation causes `Conquer The World` to be the name of a new project added to the database. Assigning that project to `@project` allows the project to be accessed from other steps, though you do need to be a little careful with that. Creating too many instance variables can kind of short-circuit the end-to-end testing that Cucumber is best at.

The way I like to work when using Cucumber is to go a step at a time. Define the step, then add any regular tests and code needed to make the step pass. (At least, that's how I like to work with Cucumber this week. It's very flexible and my work flow changes). Sometimes, this also means going back to Cucumber and changing the scenarios around some.

This step already passes. Let's move on. The next step is `And the following users`. The step definition is:

```
1 Given /^the following users$/ do |user_data|
2   user_data.hashes.each do |user_hash|
3     User.create!(user_hash)
4   end
5 end
```

Starting with maybe the most obvious point, even though the step is actually introduced in the scenario with `And`, since it's actually part of the `Given` clause, it will match against a `Given` step definition.

The body of this definition shows how to use the table data. The data comes in as a custom Cucumber object, calling `hashes` on the object converts the object to an Array of hashes, with keys corresponding to the first row of the table, and the values corresponding to each remaining row in turn. With the table that was written in the feature, this means that two users are created with the logins `alpha` and `beta`. This step passes already as well.

Moving on to the actual scenario, we start with one more given: `Given that user "alpha" is a member of the project`. This is another relatively easy one that passes without much further work.

```
1 Given /^that user "(.*)" is a member of the project$/ do |login|
2   User.find_by_login(login).projects << @project
3 end
```

To make this work we need to add `has_and_belongs_to_many` lines to both `Project` and `User` — in `app/models/project.rb`:

```
1 class Project < ActiveRecord::Base
2
3   has_many :status_reports
4   has_and_belongs_to_many :users
```

And in `app/models/user.rb`

```
1 class User < ActiveRecord::Base
2   has_and_belongs_to_many :projects
```

Okay, we've got one more that will pass before we get to the hard stuff.

```
1 When /^I visit the edit page for the project$/ do
2   visit("/projects/#{@project.id}/edit")
3 end
```

In this case, I'm using the `visit` function, provided by Webrat to simulate a browser call to the given URL. Webrat will get it's own prescription later on, for the moment all we need to know is that Cucumber lets you call Webrat functions to simulate user actions. This step also passes as is.

If you are at all inclined to be skeptical about Cucumber's general awesomeness, then right about now you're probably wondering what's the point of all this. So far, we've written some formal-sounding natural language stuff, and some Ruby that has served merely to confirm things about our application that we already know, at the cost of something like a page of code.

True so far — although to some extent this is a side effect of the fact that we're adding Cucumber tests to an application that already exists. In much the same way that adding tests to an existing legacy application has some cost, adding the first Cucumber tests to an existing non-Cucumber'd program has some cost. In essence, we're paying off some of the technical debt accrued by Huddle for not having acceptance test. Although, it's easier to add Cucumber tests to an existing program than unit tests — since Cucumber works only at the level of user input and output, any crazy or messed-up code structures in the application code don't really affect it.

As for the eventual benefit, bear with me for a little bit, I'll be making that case that after we write some successful features.

Making Step Definitions Pass

We're going to write the next three step definitions together, then make them all pass. The first step definition to write is for the step `Then I should see 2 checkboxes`. The definition of the step looks for checkboxes in the HTML using `assert_select`.

```
1 Then /^I should see (.*) checkboxes$/ do |checkbox_count|
2   assert_select "input[type = checkbox][id *= user]",
3       :count => checkbox_count.to_i
4 end
```

The next step definition to write is for the step `Then the "alpha" checkbox should be checked`. As it happens, this step is already defined by Cucumber as part of the webrat_steps.rb file.

```
1 Then /^the "([^\"]*)" checkbox should be checked$/ do |label|
2   field_labeled(label).should be_checked
3 end
```

In Webrat-ese, this means there is a checkbox field with the given label, and the field is checked — note that you really do need to have RSpec matchers hanging around for this to work.

Somewhat annoyingly, the negative version of the above test is not provided, but it's easy enough to add to our `users_and_projects_steps.rb` file:

```
1 Then /^the "([^\"]*)" checkbox should not be checked$/ do |label|
2   field_labeled(label).should_not be_checked
3 end
```

At last, we have a direction for our code, we need to add the checkboxes to the edit view for the project.

What happens at this point is a matter of style, and I haven't quite figured out where I stand yet. The question is whether to also write controller tests to cover the creation of the checkboxes on the edit

page or not. The case against is that the controller test is just a duplication of the Cucumber test. The case in favor is that code should have unit test coverage, and the Cucumber tests are decidedly not unit tests.

My rules of thumb at the moment:

- Cucumber should not replace the TDD tests you would normally write. With the arguable exception of a) "happy path" controller tests, and b) high-level user testing. That's optional, though. If it feels like there's enough logic in the controller or view to warrant writing the extra test, go for it.

- Cucumber is not the place to test internals of the program. Internal logic of models should be tested in unit tests. Internal logic in controllers should probably be moved to models. Okay, you do need to have actual logic in controllers — those should be tested in controller tests.

- Specifically, use the regular tests to determine if an object or action is valid, use cucumber to specify what the user sees on an invalid action.

- Or, to put that in a different context, internal logic in, say, a helper method probably needs a unit test. But view logic that substantially changes what the user sees can be exposed at the Cucumber level.

- If you think you can make the Cucumber step pass in the best-case length of a TDD pass/fail/refactor loop (like, about fifteen minutes), then you probably don't need another test. If making the step pass is more complex, then you probably do more tests.

- The point of Cucumber is decidedly not to get bogged down in whether you need extra tests or not. The point of Cucumber is to write better code, and focus your development efforts.

Anyway, adding the checkboxes to the edit screen probably qualifies as a "happy path" controller test. What with this being the Cucumber prescription and all, I'm feeling bold, so lets go to directly to the view.

I inserted this right below the `name` field in the file `app/views/project/edit.erb.html`. I am under no illusion that this view code will scale beyond about 10 users.

```
<h2>Users In Project</h2>
<table>
  <% User.by_login.all.each do |user| %>
    <tr>
      <td>
        <%= check_box_tag "users_in_project[]", user.id,
            user.projects.exists?(@project.id),
            :id => dom_id(user, :checkbox) %>
      </td>
      <td>
        <label for="<%= dom_id(user, :checkbox) %>">
          <%= h user.login %>
        </label>
      </td>
    </tr>

  <% end %>
</table>

```

So, for each user in the database, we add a table row with a checkbox, and user name. The `label` tag is important here, because that's what the Cucumber/Webrat steps are looking for.

At this point, the scenario passes. Yay, us! Now let's get to work on the next one.

The Edit Scenario: Specifying Paths

Since it's been pages and pages since we've seen the second scenario, I'll re-run it here:

```
1 Scenario: See users in project edit
2   Given I am on the edit page for "Conquer The World"
3   When I check "alpha"
4   And I press "Update"
5   Then I am taken to the show page for "Conquer The World"
6   And I should see "alpha@example.com"
7   And I should not see "beta@example.com"
```

This scenario currently fails right off the bat with `Given I am on the edit page for "Conquer The World"`. Hmm. `Given I am on` is a pre-existing Webrat step, but the rest of step assumes something that can be converted to a URL.

Let's try this:

```
1 Given /I am on the edit page for "(.*)"/ do |project_name|
2   @project = Project.find_by_name(project_name)
3   visit("/projects/#{@project.id}/edit")
4 end
```

Seems reasonable, if very similar to the `When` statement I wrote a couple of paragraphs back. The problem, though, is that Cucumber doesn't like it:

```
Ambiguous match of "I am on the edit page for "Conquer The World"":

     features/step_definitions/webrat_steps.rb:6:in `/^I am on (.+)$/'
     features/step_definitions/users_and_projects_steps.rb:15:in `/I am on the edit pa
```

Which, I guess, answers the question of what Cucumber does if there's a step that matches two definitions.

There's another way to specify the path. The definition for the default Cucumber step `/I am on (.*)/` defers to a method called `path_to`, which is passed the regular expression group as an argument. That method is defined in the file `features/support/paths.rb` as follows:

```
def path_to(page_name)
  case page_name
  when /the homepage/
    root_path
  when /the new users_and_projects page/
    new_users_and_projects_path
  # Add more page name => path mappings here
  else
    raise "Can't find mapping from \"#{page_name}\" to a path."
  end
end
```

See that part where it helpfully says "Add more page name => path mappings here"? What you want to do here is add more where clauses that return some kind of Rails URL like object. So, delete the `Given` clause we just wrote, and add the following `when` clause to the `path_to` method in `features/support/paths.rb`:

```
when /edit page for "(.*)"/
  @project = Project.find_by_name($1)
  edit_project_path(@project)
```

I want to walk this out step by step, because it took me a couple tries to figure this out.

1. Cucumber sees the step `Given I am on the edit page for "Conquer The World"`.

2. The step is matched to the existing Webrat step `Given /^I am on (.+)$/`.

3. The Webrat step passes the grouped text: `the edit page for "Conquer The World"` to the `path_to` method.

4. The `path_to` method matches the regular expression you just added, `/edit page for "(.*)"/`.

5. The grouped text in this regex is used to convert that expression to the RESTful edit path for the associated project.

There are two ways of looking at this mechanism. On the plus side, it's a very flexible way to allow a path to be specified in a meaningful plain language way. On the other hand, it's got a certain fog-on-fog quality, and there's a lot of indirection.

At this point the step should pass, and the next two steps, `When I check "alpha"` and `And I press "Update"`, should also pass because they are Webrat steps that are only dependent on the existence of the form elements we created for the first scenario. They have the effect of mimicking the user actions of checking a checkbox and submitting the edit form with one user checked.

The fourth step, `Then I am taken to the show page for "Conquer The World"` needs to be defined. The intent of this step is to validate that the form submission takes the user to the page intended. The step definition looks like this:

```
1 Then /^I am taken to (.*)$/ do |path|
2   assert(current_url.ends_with?(path_to(path)))
3 end
```

And is dependent on another clause in the `path_to` method, very similar to the last one:

```
when /show page for "(.*)"/
  @project = Project.find_by_name($1)
  project_path(@project)
```

The step definition is comparing the Webrat `current_url`, which will be `http://www.example.com/projects/1` to the URL path output from `path_1`, which will be `projects/1`.

Now we're at the meat of the scenario. The last two steps, `And I should see "alpha@example.com"` and `And I should not see "beta@example.com"` are Webrat steps that search for specific text in the body output. In this case, we're assuming that the eventual project show page will include the emails of the users on the project — and not include the emails of users who aren't in the project.

You have to be careful here — the Cucumber test is explicitly not testing that the user has actually been added to the project in the database. You could write such a test, but it seems to be considered better practice to manage that at the controller and model test level and keep Cucumber at the level of user interaction. That's fine, but it's also true that I could make the Cucumber test pass by just including the text string in the output. Or, more insidiously, by just passing the form submission data to the view without saving it.

My point here is not that Cucumber is bad — it's not. It's more to say that Cucumber is only part of your nutritious testing breakfast. In this case, we need to step down to controller tests. Put this controller test in `test/functional/projects_controller_test.rb`:

```
1 test "should update with users" do
2   set_current_project(:huddle)
3   put :update, :id => projects(:huddle).id,
4       :users_in_project => [users(:quentin).id]
5   huddle = Project.find_by_name("Huddle")
6   assert_equal [users(:quentin).id], huddle.user_ids
7 end
```

This confirms that sending user ids to the update will be converted to users in the actual project. The case where there are no users being passed is actually being taken care of by the already existing update test. To pass the controller test, the following goes into `update` in `app/controllers/projects_controller.rb`.

```
def update
  @project = Project.find(params[:id])
  @users = begin User.find(params[:users_in_project]) rescue [] end
  @project.users = @users
  respond_to do |format|
    if @project.update_attributes(params[:project])
      flash[:notice] = 'Project was successfully updated.'
      format.html { redirect_to(@project) }
      format.xml  { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml  { render :xml => @project.errors,
          :status => :unprocessable_entity }
    end
  end
end
```

That passes the controller test, and puts the data in the database. But to pass the Cucumber test, the data needs to get into the view. Please add the following to the beginning of the `app/views/projects/show.erb.html` file:

```
1  <h2>Users for <%= @project.name %></h2>
2
3  <table border="0" width="">
4    <% @project.users.each do |user| %>
5      <tr>
6            <td><%= h user.login %></td>
7            <td><%= h user.email %></td>
8        </tr>
9    <% end %>
10 </table>
11
```

I will grant that's not anything like a beautiful design. But Cucumber doesn't care, and happily passes. All green, the feature is complete, for some definition of complete.

Login And Session Issues

There's a big issue that I've kind of glossed over so far — authentication. In a real application, you'd probably need to be logged in to view and edit projects. From a controller test, simulating a login is easy — we have direct access to the controller and session, so you can add the fake user directly to the fake session.

From Cucumber, we don't have access to the controller or the session, so we need to login by simulating user actions. Something like this works if you are using RESTful Authentication.

```
Given /^I am logged in$/ do
  User.delete_all
  @user = User.create!(:user,
      :login => "the_login",
      :password => "password",
      :password_confirmation => "password")
  visit "/login"
  fill_in("login", :with => "the_login")
  fill_in("password", :with => "password")
  click_button("Log in")
end
```

This step definition creates a user for as an instance variable, then travels to the RESTful authentication login screen and simulates filling in and submitting the login form with Webrat. If we wanted a specific user name, it'd be easy enough to change the step definition to take one in. On the plus side, this is largely boilerplate and can be applied to most projects with little or no modification.

Anything session-based will have a similar issue, and will need to be generated by simulating user actions. Again, this is means that Cucumber isn't going to solve all your testing problems. Just some of them.

Concluding Thoughts

For a long time, Cucumber and it's predecessor StoryRunner were on my list of interesting projects that I might get around to someday, but which didn't seem tremendously practical. One of the things that made me turn the corner and start using it was the realization that a lot of my objections to Cucumber —

writing extra code, the fact that the tests might not always be accurate, that kind of thing — were very similar to the kinds of objections I'm always hearing about Test-Driven Development in general. That gave me pause, and I decided to make a concerted effort to try and figure Cucumber out.

As I write this, I've been basically obsessed with Cucumber for a few weeks — adding it to any project I touch, presenting it to co-workers, writing about it. It's no substitute for years of experience or anything, but I think I'm getting a sense of how to use Cucumber effectively and how to get really burned by it.

Some thoughts:

- From a developer's perspective, it seems like the most useful way to think of Cucumber is as a domain-specific language for specifying integration and (to a lesser extent) view tests. The natural language syntax makes it easy for a development team to agree on what the tests mean, and give the Cucumber tests a somewhat greater chance of being useful. The value of having relatively easy end-to-end testing is potentially pretty high, it's not at all uncommon for me to see bugs get into an application due to the lack of coordination between unit and controller and view tests.

- If you are dealing with a client or non-programming customer, the value of an acceptance test that is both natural language and executable is outstanding. My only caution at the moment would be that even though there isn't much structure in Cucumber tests, there still is some. We tend to use the `Given/When/Then` structure for user stories even without Cucumber, but my first attempts at translating some of them to Cucumber still required some translation. For example, the story definitions tended to do too much in one scenario.

- There is absolutely no doubt that there is an up front cost in starting to write Cucumber tests, especially coming up to speed on an existing project. There's a cost in writing the step definitions, and there's a cost in maintaining the correctness of the tests over time. I don't think it's an

insurmountable cost, and I think there are many ways that it will pay for itself over time, but there's definitely a start-up cost.

- It seems clear that the key to whether you have a good experience or a bad experience over time with Cucumber is how well you write the step definitions. Right now I would argue for a) let the scenario have the natural language that feels right, b) keep the step definitions very simple, c) resist the temptation to have many regular expression groups in the same step definition to make it match more steps — it's fine to have multiple simple steps, d) keep the `When` and `Then` steps at the level of the user, rather than the database, e) as with other tests, verifying what isn't there is as important as verifying what is, f) Cucumber is not the place to sweat details, save that for the regular tests, and g) be very careful to avoid tautological tests — it seems like it'd be pretty easy for the indirections of the step definitions to mask an always-passing step.

- Also, be prepared to go back and forth and rewrite your Cucumber steps a little bit over time for clarity and because new features become apparent.

- I'm a little concerned about how Cucumber would play out on a large project. Not so much the performance, as the increased possibility of naming crashes in the step definition space. Even with just one feature plus the Webrat steps, I still was having some trouble keeping all the steps straight.

- Cucumber is somewhat limited — right now — in handling Ajax. For one thing, Webrat does not, as I write this, follow rails `link_to_remote` links properly. For another, the Webrat/Cucumber matchers for responding to Ajax results don't quite work — although I think you can still use the Rails `assert_select_rjs` for some things. Various kinds of behavior triggered by focus changes or mouse over is going to be awkward to test.

- It seems like the pattern in the two scenarios in this section — one to verify the display, and others to validate user actions — is the way to start when working on a new feature via Cucumber.

- Another great use of Cucumber is in doing black box testing around a legacy project. One of the problems with writing tests on a project that has gone a long time without them is that normally you need to refactor in order to get useful tests, but you don't have tests to tell you if your refactor breaks anything. With Cucumber, you can write elaborate tests about the behavior of the system without touching the application code, but you still have some protection against breaking something when writing unit tests.

I mentioned that Cucumber would probably recover its up-front costs over time. How might that happen?

- In a client situation, if agreeing on Cucumber requirements saves even one email back-and-forth, you're well on your way to recovering the time.

- Even in a developer-only situation, Cucumber is a great way to figure out what the user interaction is going to be like. I found it to be helpful in focusing what I wanted to do and guiding development from step to step.

- I suspect that using Cucumber to drive development will keep you from spending time working on unneeded features.

Acknowledgements

A number of people have helped in the creation of this text.

Several people commented on an early version of the layout of this book. Brandon Martinez was good enough to create a complete design that was, shall we say, influential in changes I made. Somebody named Mileszs was kind enough to point me at CodeRay.

Early readers of this text who provided useful comments included Paul Berry, Anthony Caliendo, Brian Dillard, Sean Hussey, John McCaffrey, Matt Polito, Christopher Redinger.

Pathfinder Development has been very helpful in hosting the Rails Prescriptions site and providing other support. Special thanks to Bernhard Kappe, Deitrich Kappe, Alan Choyna, and David DiGioia. Alice Toth provided the original design for the Rails Prescriptions website.

Other readers who made valuable corrections to this book over time include Eric Cranston, Sigfrid Dusci, and John Yerhot.

Dana Jones has been made a number of valuable editorial corrections to the book, and has been a strong online salesperson besides.

Colophon

This text was put together using PrinceXML. The original text was written in Markdown, and PrinceXML was used to convert to the PDF using a series of scripts based on work done by Carlos Brando. The CSS template used was based on an original style for boom!, the book microformat written by Hakon Wium Lie and Bert Bos. Syntax coloring and line numbering is done using CodeRay.

The body font is 12 point Century Gothic, the code font is 12 point Monaco.

Changelog

March 13, 2009

- Initial Public Release

March 15, 2009

- Fix to status date test in getting started section, found by Dalibor Nasevic

April 6, 2009

- Addition of Cucumber and Autotest sections
- More content in the Style prescription

April 20, 2009

- New instructions for registering content
- The very beginnings of a chapter on Shoulda

May 3, 2009

- Completed Shoulda chapter
- New chapter on working with legacy systems

- New cover

May 25, 2009

- Long chapter on RSpec
- Messed with the cover some more
- Many typos fixed, courtesy of Dana Jones.

Appendix: Application setup

What we need to do first is get the Rails app set up with RESTful Authentication. You'll need SQLite3 for this, and if you want to follow along with every step of the example, it will help to have git installed.

I want to run the application against Edge Rails 2.3, but only have gem Rails 2.2 loaded on my machine, which requires me to go get the Rails edge code then rerun the `rails` command:

```
noel$ mkdir huddle
noel$ cd huddle
noel$ rails .
noel$ rake rails:freeze:edge
noel$ rails .
noel$ rake db:create:all
```

I've seen at least one report that `rake rails:freeze:edge` doesn't work on Windows. If you have a problem, you can download the latest Rails as a zip file directly from GitHub at http://github.com/rails/rails/tree/master

Then, load up RESTful authentication.

```
noel$ git clone git://github.com/technoweenie/restful-authentication.git
  vendor/plugins/restful_authentication
noel$ rm -rf vendor/plugins/restful_authentication/.git
noel$ script/generate authenticated user sessions
noel$ rake db:migrate
```

Next up, put everything in a git repository.

```
noel$ git init
noel$ mate .gitignore
noel$ git add .
```

A couple of program notes before I get way too far in the weeds here. The `.gitignore` file I'm using looks like this:

```
log/*
doc/*
tmp/*
.DS_Store
.rake_tasks~
```

I'm not keeping the Rails or RESTFul Authentication in a separate git repository so I can keep everything all together, it's easier to manage as a code sample that I'm going to distribute that way.

Now, to get all the way in the weeds, as I write this in Edge Rails (2.3), you need to change the class definition in the `test/test_helper.rb` file from

```
1 class Test::Unit::TestCase
```

to

```
1 class ActiveSupport::TestCase
```

You may also need to change the file name `app/controller/application.rb` to `app/controller/application_controller.rb` depending on the version of Rails that you used to generate the initial application

```
mv app/controllers/application.rb app/controllers/application_controller.rb
```

Then you can commit back to git.

```
noel$ git commit -am "starting point"
```

Note that this is creating SQLite versions of the database, that's fine for now, in the interest of having it be as easy as possible to run this tutorial.

We already have a user module and partial user controller generated by RESTful authentication. We'll need a model for the status reports and one for the project. Right now, a project is just a string:

```
noel$ script/generate scaffold project name:string
```

And a status report has references to project and user, plus text fields for the yesterday and today reports. I've also added an explicit date for the status, separate from the `created_at` field.

```
noel$ script/generate scaffold status_report project:references
    user:references yesterday:text today:text status_date:date
```

Projects and users have a many-to-many relationship, so you'll need a join table for that:

```
noel$ script/generate migration project_user_join
```

The migration looks like this:

```
1  class ProjectUserJoin < ActiveRecord::Migration
2    def self.up
3      create_table :projects_users, :force => true, :id => false do |t|
4        t.references :project
5        t.references :user
6        t.timestamps
7      end
8    end
9
10   def self.down
11     drop_table :projects_users
12   end
13 end
```

Then run `rake db:migrate`.

The First Tests

That's a long way to just to get started. Sorry, it's the kind of set up you have to get out of the way to do any kind of example complex enough to be useful.

It's worth pointing out, though, that we already have some tests. Running the default `rake` testing task gives output like this. (Output slightly truncated):

```
noel$ rake
(in /Users/noel/Projects/huddle)
Started
..............
Finished in 0.24366 seconds.

15 tests, 28 assertions, 0 failures, 0 errors

Started
...........................
Finished in 0.596605 seconds.

28 tests, 46 assertions, 0 failures, 0 errors
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/bin/ruby
    -Ilib:test "/Library/Ruby/Gems/1.8/gems/rake-0.8.3/lib/rake/rake_test_loader.rb"
```

The first batch is the unit tests — Rails creates a dummy test for each model, plus RESTful Authentication defines tests. The bottom tests are the functional tests. Rails creates a basic suite for each controller scaffold that covers the successful cases of each scaffold method. (The failure cases can easily be

covered with mock objects). And again, the UsersController and SessionsController classes are covered partially by RESTful Authentiation.

This brings us to the git branch `step_1`, and you can rejoin the tutorial at the beginning.