Testing and Plugins

INFO 2310:
Topics in Web Design and Programming

Matt Kulick '07, Associate Product Manager at Google, is coming!

ISSA General Meeting with Matt Kulick '07
IS @ Cornell and beyond
Tuesday, 10/21 @ 4:30 PM, Upson 205

Google Tech Talk with Matt Kulick '07
Wednesday, 10/22 @ 5:00 PM, Phillips 101

infoscience STUDENT ASSOCIATION

Where we were

Last time we started talking about testing our Rails code. Recall that there are three types of tests:

- Unit tests (for models)
- Functional tests (for single controllers)
- Integration tests (for everything together)

Baked in

Rails has support for this already built-in in the test directory. Note there are four directories: test/unit, test/functional, test/integration, and test/fixtures.

Recall that fixtures let us define test data for our tests.

More about fixtures

More DRYness

Fixtures allow you to define data once and reuse it in multiple tests. For instance, let's create some sample users more imaginative than the defaults.

Add this to test/fixtures/users.yml:

```
<% salt = Digest::SHA256.hexdigest('salt') %>
<% pass = Digest::SHA256.hexdigest("--#{salt}--password--")%>

matz:
  name: Yukihiro Matsumoto
  email: matz@ruby-lang.org
  salt: <%= salt %>
  crypted_password: <%= pass %>

why:
  name: Why the Lucky Stiff
  email: why@whytheluckystiff.net
  salt: <%= salt %>
  crypted_password: <%= pass %>
```

Note that we can use ERB in these yml files…

Associations

We can now have other fixtures refer to these ones.
Edit test/fixtures/posts.yml:

```
one:
  title: My First Post
  body: Hurray!
  author: matz

two:
  title: Another Post
  body: Here is another post.
  author: why
```

More tests

Now we can write tests referring to users in test/unit/post_test.rb.

```
def test_should_be_editable_by_its_author
  assert posts(:one).editable_by?(posts(:one).author)
end
```

Try to write test_should_not_be_editable_by_a_non_author

Running unit tests

Recall that you can run unit tests either through Komodo (Rails Tools/Test/Unit Tests) or via rake (‘`rake test:units`’).

Functional tests

Functional tests

We test the controllers via functional tests. Note that the scaffold created some reasonable functional tests in test/functional/posts_controller_test.rb such as

```
def test_should_get_index
  get :index
  assert_response :success
  assert_not_nil assigns(:posts)
end
```

We can run these via ‘`ruby –I test test/functional/posts_controller_test.rb`’. We could also run ‘`rake test:functionals`’, but this will bomb bigtime…

Problems

In fact, even here we get two problems.

The first is with creating a post, which fails. Why?

The second is with editing a post, which has an error. Why?

First fix

The first is an error in test_should_create_post since a post with no title and body is created.

```
def test_should_create_post
  assert_difference('Post.count') do
    post :create, :post => {:title => 'A post',
  :body => 'A body' }
  end

  assert_redirected_to post_path(assigns(:post))
end
```

Second problem

In a sense, the second issue shows that things are working the way they are supposed to – but we need to test them differently.

In this case, we need to have the ability to log in a user. So let's write a helper function to do this.

In test/test_helper.rb, add (at the bottom):

```
def login_as(user)
  @request.session[:user_id] = user
  ? users(user).id : nil
end
```

Log in first

test/functional/posts_controller_test.rb

Now we can edit test_should_get_edit to login first:

```
def test_should_get_edit_if_editable_by
  login_as(:matz)
  get :edit, :id => posts(:one).id
  assert_response :success
end
```

Testing for bad guys

We can also add a test that checks to see that the exception is raised if someone does what they aren't supposed to:

```
def test_should_be_forbidden_if_not_editable_by
  login_as(:why)
  assert_raise
  AuthorizationFu::Exceptions::SecurityTransgress
  ion do
    get :edit, :id => posts(:one).id
  end
end
```

Now rerun tests (‘ruby –I test test/functionals/posts_controller_test.rb’).

Integration tests

Testing logins

But suppose we *really* wanted to test the login process; that is, test the process of someone entering their login name and password, and being redirected to the right place.

For this (among many other things), we can write integration tests.

Not scaffolded

Integration tests are not created automatically upon scaffolding a model or generating a controller, so we have to generate them ourselves.

'`ruby script/generate integration_test login`' will create a file test/integration/login_test.rb.

login_test.rb

```
require 'test_helper'
class LoginTest <
  ActionController::IntegrationTest
  # fixtures :your, :models

  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

Adding fixtures

We'll want to use the user fixtures, so uncomment the fixture line and have it read '`fixtures :users`'.

A login test

```
def test_login_known_user
  get login_path
  assert_response :success
  assert_template "sessions/new"
  post sessions_path, :email =>
  users(:matz).email, :password => 'password'
  assert_response :redirect
  follow_redirect!
  assert_response :success
  assert_template "posts/index"
  assert_select "p", /Yukihiro Matsumoto/
end
```

Another example

We can also create integration tests to simulate multiple users interacting with the system at the same time.

We can use a command '`open_session`' to create a session for each user using the application.

Suppose one user deletes a comment at the same time another user is editing it.

A possible test (with no asserts)

```
def test_destroy_post_while_editing
  matz = open_session
  why = open_session
  post = posts(:one)

  # Login both users
  matz.post sessions_path, :email => users(:matz).email, :password =>
    'password'
  why.post sessions_path, :email => users(:why).email, :password =>
    'password'

  # Matz goes to edit the post
  matz.get edit_post_path(post)
   # _why decides to delete the post
  why.delete post_path(post), :id => post.id
  # Matz tries to save his edits
  matz.put post_path(post), :post => {:title => "New Title", :body =>
    "New body" }

end
```

More DRYness

We could also make this much more DRY by writing lots of helper methods, so that the test would look something like:

```
def test_create_edit_comment
  matz = open_session_as(:matz)
  why = open_session_as(:why)
  topfunky = open_session # not a registered user

  new_post = matz.creates_post(...)
  why.edits_post(new_post, ...)
  topfunky.comments_on(new_post, ...)
end
```

DSLs

Note that you are close to writing your own little testing language in this case; sometimes called a *domain specific language*, and makes it easy to write lots of tests in a compact way…

Other stuff on testing

There are various other testing frameworks to help you write tests. Dean mentions:

- Shoulda (http://www.thoughtbot.com/projects/shoulda) (alternate unit testing)
- Factory Girl (http://www.thoughtbot.com/projects/factory_girl) (alternative to fixtures)

```
class UserTest < Test::Unit
  context "A User instance" do
    setup do
      @user = User.find(:first)
    end

    should "return its full name" do
      assert_equal 'John Doe', @user.full_name
    end

    context "with a profile" do
      setup do
        @user.profile = Profile.find(:first)
      end

      should "return true when sent #has_profile?" do
        assert @user.has_profile?
      end
    end
  end
end
```

Produces the following test methods:

```
"test: A User instance should return its full name."
"test: A User instance with a profile should return true when sent #has_profile?."
```

TATFT

Also, an amusing, slightly jargony, but pointed talk on testing:

http://rubyhoedown2008.confreaks.com/05-bryan-liles-lightning-talk-tatft-test-all-the-f-in-time.html

Gems and Plugins

Extensibility

One of the nice things about Rails is that there are a number of prepackaged bits of code created by other users that you can incorporate into your own code.

They come in two flavors: gems and plugins.

Gems

RubyGems is the Ruby package management system. So not always specific to Rails; they just happen to be useful.

Recall that Rails itself is a gem.

To use them, we list them specifically in a file in config/environment.rb.

Plugins

Plugins are specific to Rails. They are present in vendor/plugins, and are automatically loaded when the server is started.

Plugins seem to be on the wane in Rails. Gems used to be more of a pain to deal with, but a part of the new version of Rails helps deal with gem dependencies, which has lessened the need for plugins.

Using a gem

Maruku

To give ourselves some ideas of how a gem gets used, we'll practice with "Maruku", a gem that interprets 'markdown' (see http://maruku.rubyforge.org/ and http://en.wikipedia.org/wiki/Markdown).

Including Maruku

To tell Rails we want to use Maruku in config/environment.rb, we add a line saying this. Look for the comment "`# Specify gems this application depends on`". After the comment, add the line

```
config.gem 'maruku'
```

Rake

Now we can use some rake-related tasks to help us manage our gems. Try typing '`rake –T gems`'.

Now try '`rake gems`'.

Now '`rake gems:install`'.

Now '`rake gems:unpack:dependencies`'. This will put the gem in your blog directory (and any gems it depends on), so we don't need to rely on it being installed on the machines in this cluster. In general you might not need to do this if the gem is installed on the machine you're using.

Using Maruku

In views/posts/show.html.erb, we can edit

```
<p>
  <b>Body:</b>
  <%=h @post.body %>
</p>
```

to

```
<p>
  <b>Body:</b>
  <% body = Maruku.new(@post.body) %>
  <%= body.to_html %>
</p>
```

Now try it…

Fire up the server, and try a post like this…

```
# This is the headline!

Now for an *itemized* list:
- First item
- Second item

And some code:
`<%= body.to_html %>`
```

Using a plugin

A nice thing

There are lots of plugins: there's a huge directory of them at http://agilewebdevelopment.com/plugins.

Git

It's useful to do this via version control software (such as Git). This is slightly problematic in that Git isn't currently installed on these machines.

To install on your own machine, go to http://code.google.com/p/msysgit/downloads/list, and install the first program listed ("Full installer if you want to use official Git 1.6.0.2").

will_paginate

To give ourselves some practice using a plugin, we'll use the plugin 'will_paginate', something that let's us paginate a list of items (so that only X of them appear at a time).

Installing will_paginate

If you have Git installed, you can install the plugin via the following command:

```
ruby script/plugin install
  git://github.com/mislav/will_pagin
  ate.git
```

Note that this grabs the source code from some location (github.com here) and installs it under vendor/plugins.

Installing will_paginate

If you don't have Git installed, you can download the plugin from http://github.com/mislav/will_paginate/tree/master, unzip it, and put files into vendor/plugin/will_paginate.

Using will_paginate

We now alter the index method of the posts controller to return only a subset of the posts. Rather than:

```
def index
  @posts = Post.find(:all)
```

we instead use

```
def index
  @posts = Post.paginate :page => params[:page],
  :order => 'updated_at DESC'
```

Using will_paginate

This may not do much yet, since the default is 50 items per page. Let's lower the default:

```
def index
  @posts = Post.paginate :page =>
  params[:page], :per_page => 5,
  :order => 'updated_at DESC'
```

Using will_paginate

We also need to add pagination controls to our view. At the bottom of views/posts/index.html.erb, add:

```
<%= will_paginate @posts %>
```

before

```
<%= link_to 'New post',
  new_post_path %>
```

Try it!

You can experiment with the ordering, number of items per page, etc.

For more info, look at http://github.com/mislav/will_paginate/tree/master.

Gems vs. plugins

Nice things about gems:

- If installed in the underlying system, you don't need to have them in your repository (unlike plugins)
- Can easily update them all (via `gem update`). Some work involved in keeping plugins up-to-date.
- Deals nicely with dependencies on other gems.

Nice things about plugins:

- There are still a lot of them that aren't gems yet.

More fun things…

Other plugins worth looking at for our blog (as recommended by Dean):

- Add tags to posts using has_many_polymorphs (http://blog.evanweaver.com/files/doc/fauna/has_many_polymorphs/files/README.html).
- Let people leave comments via Open ID authentication (http://github.com/rails/open_id_authentication/tree/master)
- Attach images to uploads via paperclip (http://www.thoughtbot.com/projects/paperclip).
- Allow full-text searching using ultrasphinx (http://blog.evanweaver.com/files/doc/fauna/ultrasphinx/files/README.html).

Reminders

I do expect people to do more with their blog than simply what we did in class, by adding styling or functionality in some way. Gems/plugins might be a good way to go about this.

Matt Kulick ’07, Associate Product Manager at Google, is coming!

ISSA General Meeting with Matt Kulick ’07

IS @ Cornell and beyond

Tuesday, 10/21 @ 4:30 PM, Upson 205

Google Tech Talk with Matt Kulick ’07

Wednesday, 10/22 @ 5:00 PM, Phillips 101

infoscience STUDENT ASSOCIATION