

Authorization and Testing

INFO 2310:
Topics in Web Design and
Programming

Where we are

Last time: We added the ability of users to log in, and made sure that we treated their passwords in a secure fashion.

This time: How can we use user information to limit actions?

For example

Wouldn't it be nice to ask a post whether the current user is authorized to edit it?

```
@post.editable_by?(current_user)
```

We could further add methods to correspond to each of the possible CRUD (Create/Read/Update/Destroy) operations:

```
viewable_by?  
creatable_by?  
editable_by?  
destroyable_by?
```

Try it...

We can just add this to our post model in `app/models/post.rb`.

```
def editable_by?(editor)  
  editor == self.author  
end
```

Then in the `views/posts/show.html.erb`, we can restrict access to editing. Change

```
<%= link_to 'Edit', edit_post_path(@post) %>
```

to

```
<%= link_to 'Edit', edit_post_path(@post)  
  if @post.editable_by?(current_user) %>
```

Do the same in `views/posts/index.html.erb`.

Extensions

Now we can add the same methods to User and Comments...

But isn't this a little un-DRY? (Recall DRY = 'Don't Repeat Yourself!') How can we add default methods to all these objects at once?

The magic of open classes

All our models are subclasses of ActiveRecord::Base. So we can just add our desired extra methods to this object... which Ruby allows us to do. Then all the models will have these by default.

Message passing


A bit about how Ruby method calls work: they all pass messages to the object. So

```
object.explode!  
is actually passing the message  
:explode! to the object. We can also  
do this directly by writing  
object.send(:explode!)
```

A module for the default methods

We create the default methods in an 'AuthorizationFu' module, which we will save as lib/authorization_fu.rb. The final line instructs ActiveRecord to include the methods as part of the class.

```
module AuthorizationFu  
  module InstanceMethods  
    def viewable_by?(viewer)  
      true  
    end  
    def creatable_by?(creator)  
      true  
    end  
    def editable_by?(editor)  
      true  
    end  
    def destroyable_by?(destroyer)  
      true  
    end  
  end  
end  
ActiveRecord::Base.send(:include, AuthorizationFu::InstanceMethods)
```



Loading the methods

Add a require to the end of config/environment.rb to force this to get loaded on initialization (note that you need to restart the server to get this to work).

```
require 'authorization_fu'
```

Almost

This almost works to restrict editing access to the post author. What have we overlooked?

Remember that people can navigate directly to the editing page /posts/3/edit. How can we prevent this?

Raising an exception

Let's raise an exception in the controller if someone tries to edit a post who isn't suppose to.

In app/controllers/posts_controller.rb, in the edit action, after finding the post, add

```
raise SecurityTransgression unless  
  @post.editable_by?(current_user)
```

Adding an Exception

Now we need to add `SecurityTransgression` as a subclass of the `Exceptions` class, but we can do this in our `lib/authorization_fu.rb` file:

```
module AuthorizationFu
  module InstanceMethods
    ...
    module Exceptions
      class SecurityTransgression < Exception
      end
    end
  end
end
```

```
ActiveRecord::Base.send(:include,
  AuthorizationFu::InstanceMethods)
ActionController::Base.send(:include,
  AuthorizationFu::Exceptions)
```

Recovering gracefully

Rather than blowing up with an exception whenever someone tries to do this, we should just give them a 403 error. So we need to catch them somewhere.

We can do this with a method `rescue_action` defined in `app/controllers/application.rb`. This will catch any exception raised in any of the controllers.

To the rescue

Add to `app/controllers/application.rb`:

```
def rescue_action(e)
  case e
  when SecurityTransgression
    respond_to do |format|
      format.xml { head :forbidden }
      format.html { render :file =>
        "#{RAILS_ROOT}/public/403.html", :status =>
          :forbidden }
    end
  else
    super(e)
  end
end
```

403

Note that this renders the `403.html` file found in `public`. Except there isn't one. So take the 404 page and make one.

Debugging

Rails has a breakpointing/debugging ability. The server can be started with a debugging option via `'ruby script/server -u'`. Then any place the `'debugger'` command occurs in your code will drop you into a debugger.

You need to install the debugger first via 'gem install ruby-debug'. Note that this hasn't been done on the machines in this cluster.

Testing

You're testing, right?

Rails *assumes* you will be writing tests to check your code. There is a separate test version of the database, and default test folder. When we created our post, user, and comment models via scaffolds, Rails automatically created test files to go along with everything else.

Test types

Rails supports three different types of tests:

- Unit tests: For models
- Functional tests: For controllers
- Integration tests: For everything altogether

Note that within the test directory, there are unit, functional, and integration folders (also a fixtures folder for creating test data).

Fixtures

Some sample data from test/fixtures/user.yml:

```
one:
  name: MyString
  email: MyString
  password: MyString

two:
  name: MyString
  email: MyString
  password: MyString
```

Our users don't have passwords any more, so delete these lines.

A starting point

Let's look at test/unit/post_test.rb:

```
require 'test_helper'

class PostTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

Unit tests

Rails tests are built on top of Ruby's Test::Unit framework. We write tests called `test_blahblahblah` and assert some number of conditions.

If all conditions are true, the test passes, otherwise it fails.

Running tests

We usually run tests via rake. To see what we can run from rake, try `'rake -T test'`.

```
rake db:test:clone          # Recreate the test database from the current...
rake db:test:clone_structure # Recreate the test databases from the develo...
rake db:test:prepare       # Prepare the test database and load the schema
rake db:test:purge         # Empty the test database
rake test                  # Run all unit, functional and integration tests
rake test:functionals    # Run tests for functionalsdb:test:prepare / ...
rake test:integration      # Run tests for integrationdb:test:prepare / ...
rake test:plugins          # Run tests for pluginsenvironment / Run the ...
rake test:recent           # Run tests for recentdb:test:prepare / Test ...
rake test:uncommitted      # Run tests for uncommitteddb:test:prepare / ...
rake test:units        # Run tests for unitsdb:test:prepare / Run th...
```

Let's try it!

For this lesson, we'll focus on unit testing. So let's try to run unit tests via `'rake test:units'`. (From Komodo: 'Rails Tools/Test/Unit Tests').

```
Started
...
Finished in 2.532 seconds.

3 tests, 3 assertions, 0 failures, 0 errors

We didn't write any tests yet! What ran?
```

What should we write?

What kind of tests should we write? Here are some starting suggestions:

```
test_should_be_valid_with_valid_attributes
test_should_be_invalid_without_a_title
test_should_be_invalid_without_a_body
test_should_be_creatable_by_any_user
test_should_be_editable_by_its_author
test_should_not_be_editable_by_a_non_author
```

Let's try some!

Write the following tests in `test/unit/post_test.rb`, then run them.

```
def test_should_be_valid_with_valid_attributes
  assert_valid Post.new(:title => 'Test Post', :body =>
    'Test body.')
end

def test_should_be_invalid_without_a_title
  assert Post.new(:title => nil, :body => 'Test
    body.').valid?
end

def test_should_be_invalid_without_a_body
  assert Post.new(:title => 'Test Post', :body =>
    nil).valid?
end
```

Some issues

What happens if we end up adding some new field to the post model?

In order to avoid rewriting all our tests every time this happens, we write a helper function to create a new post, such that we can override the fields as needed.

Add the following to test/unit/post_test.rb at the bottom:

```
private
def new_post(params = {})
  Post.new({:title => 'Test Post', :body
=> 'Test body.'}.merge(params))
end
```

Now rewrite your tests to take advantage of this.
E.g.

```
def
  test_should_be_valid_with_valid_attr
  ibutes
  assert_valid new_post
end
```

Try them again...

Another issue

Another minor issue is that since Rails doesn't have an 'assert_invalid' helper function, we have to resort to 'assert !Model.valid?'. Let's fix this by adding 'assert_invalid'.

test_helper

The right place to add this is test/test_helper.rb, so that it is then available to all tests.

```
def assert_invalid(record, message=nil)
  full_msg = build_message(message, "<?> is
valid.", record)
  assert_block(full_msg) { !record.valid? }
end
```

Now rewrite (and rerun) your tests again to take advantage of the new method.

Testing philosophy

Why write tests?

- So you don't have to manually test your application every time you make a change.
- So you can be sure that a minor change in this part of the code doesn't break that part over there.
- So that when you refactor (i.e. rip up and rewrite) your code, you can be sure that the new code has the same behavior as the old code.
- So that you can CYA if making changes that affect other people's code ("It can't be broken; all your unit tests passed!")

Test-driven development

In fact, some developers make the argument that you should *start* your coding by writing the tests.

- Figure out what the code is supposed to do and write the tests to make sure it does that.
- Then write the code.
- When the tests pass, stop.

This forces you to think about what the code should do before you write the code.

Reminders...

**HAVE A GREAT
FALL BREAK!**