Validation and Testing

Siddharth Kaza
Towson University, Computer and Information Sciences

Validating input to the web application

- Why is it important?
  - Validations are used to ensure that only valid data is saved into your database.

- Where should we do it? Why?
  - Database constraints and/or stored procedures make the validation mechanisms database-dependent and can make testing and maintenance more difficult.
    - Do it if multiple applications are using the DB
  - Client-side validations can be useful, but are generally unreliable if used alone.
    - User may switch off javascript
    - But, useful because it is instant feedback to the user (not return trip to server)
  - Model-based
    - Model-level validations are the best way to ensure that only valid data is saved into your database.
    - They are DB agnostic and can be tested.

Validation

- What are the validations that we should think about in the depot application?
  - Lets run the products side of the depot and enter something in the products table
- models/product.rb

```
validates :title, :description, :image_url,
:presence=>true

validates :price, :numericality=>
{:greator_than_or_equal_to => 0.01}

 validates :title, :uniqueness => true

validates :image_url, :format =>
     { :with=> %r{\.(gif|jpg|png)$}i,
     :message => "must be an image"
     }
```

End of line

Ignore case

When are validations called?

- Most active_record methods used commonly trigger validations:
  - create, create!
  - save, save!
  - update
  - update_attributes, update_attributes!

- You can explicitly switch off validates in the save
  - save (:validate=>false)
  - When would you use this?

Validations (cont.)

- An excellent source: http://lmgtfy.com/?q=rails+validations
- Lets see the various helpers (like :presence, :numericality) available to us
  - Look closely at :format, :length, and "custom validations"
- Validations get checked when a record is saved or updated.
- When validations fails, errors are added to the error hash associated with the model.
  - This hash is used in the view to show the errors (as in Lab)

Testing

- What is automated testing?
- Why is it important in applications (and web applications)?
- How does one do it?

Testing – the old way

- Write code first, then test
- Hope that you find bugs
- “Engineer's induction” - it worked the first few times, so it must work on all cases?
- Little language/system support
  - At least little that is mentioned in textbooks.

What's wrong with this picture?

- Hard to run tests after you have completed the code
  - You think of all the options while you are coding.. May forget later
- Hard to re-run tests after changes
- Testing is easily neglected and poorly documented
  - Project gets delayed and testing gets cut (classic result of waterfall strategy)
- Code quality is lower
  - Updates, service packs, versions …

Testing, the new way

- Integrated tools for running and executing tests
- Write tests as soon as you write code
- Test early and often
- Tests become self-documenting part of code.
- More testing code than "real code"? 3:1 ?
  - Make sure that the management has bought in to the idea of test-based development.
    - else, they will wonder what you have been coding for so long?

Types of tests

- Functional tests
  - Testing the functionality of the classes/modules
  - In Rails (slightly different nomenclature):
    - Unit tests: test the models
    - Functional tests: test the controller
- Integration tests
  - Testing the interface between various classes/sub systems
  - Rails: integration tests
- System tests
  - Test the system to make sure it satisfies requirements

Rails and tests

- Guide at http://guides.rubyonrails.org/testing.html
  - Some examples used here
- When you create a Rails application it creates a test folder for you

```
$ ls -F test/

fixtures/       functional/     integration/    test_helper.rb  unit/
```

Lets look at a simple Rails application

```
$ rails generate scaffold post title:string body:text
...
create  app/models/post.rb
create  test/unit/post_test.rb
create  test/fixtures/posts.yml
...
```

- The default test stub looks like this:

```ruby
require 'test_helper'

class PostTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

- *assert* is a method that tells the test framework what we expect to be true
  - If it is not true then the test fails, else all is fine
  - Thus, each test file is a set of assertions that you expect to be true about your code at all times

Some tests on product

should be true

```
test "product attributes must not be empty" do
  product = Product.new
  assert product.invalid?
  assert product.errors[:title].any?
  assert product.errors[:description].any?
  assert product.errors[:price].any?
  assert product.errors[:image_url].any?
end
```

- We are trying to create an empty product
  - product.invalid? checks if the product object is valid or not
    - it should not be, since we had a bunch of validations (for title etc.) in there
  - product.errors… returns true if the attribute has an error associated with it
    - Again, it should since we put it in the model

We can also test individual columns/fields

```
test "product price must be positive" do
  product = Product.new(:title       => "My Book Title",
                        :description => "yyy",
                        :image_url   => "zzz.jpg")
  product.price = -1
  assert product.invalid?
  assert_equal "must be greater than or equal to 0.01",
    product.errors[:price].join('; ')

  product.price = 0
  assert product.invalid?
  assert_equal "must be greater than or equal to 0.01",
    product.errors[:price].join('; ')

  product.price = 1
  assert product.valid?
end
```

Join all the errors in the array to string

Testing image_url

arrays

```
def new_product(image_url)
  Product.new(:title       => "My Book Title",
              :description => "yyy",
              :price       => 1,
              :image_url   => image_url)
end

test "image url" do
  ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
           http://a.b.c/x/y/z/fred.gif }
  bad = %w{ fred.doc fred.gif/more fred.gif.more }

  ok.each do |name|
    assert new_product(name).valid?, "#{name} shouldn't be invalid"
  end

  bad.each do |name|
    assert new_product(name).invalid?, "#{name} shouldn't be valid"
  end
end
```

Fixtures

- Fixtures provide a way to assume initial data in the test database
  - much like the migrations in development/production database
  - but better since fixtures are run before the start of *each* test method

- look at the test/fixtures directory
  - each row is given a name
  - make sure you use spaces to indent
  - give each record a name (if you so wish)

- They have their own directory as they will be needed for testing other functionality too (as we shall soon see).

Fixtures (cont.)

- Rails by default automatically loads **all fixtures** from the test/fixtures folder for your unit and functional test.

- Loading involves three steps:
  - Remove any existing data from the table corresponding to the fixture
  - Load the fixture data into the table
  - Dump the fixture data into a variable in case you want to access it directly

Testing if the title uniqueness validation works

```
test "product is not valid without a unique title" do
  product = Product.new(:title       => products(:ruby).title,
                        :description => "yyy",
                        :price       => 1,
                        :image_url   => "fred.gif")

  assert !product.save
  assert_equal "has already been taken", product.errors[:title].join('; ')
end
```

Running the test (command prompt)

- Notice Rails has three development environments
  - Production, development, test
  - Each has its own DB.

  - rake test:units

Running the tests in the product unit tests

```
[vmuser@fedora14 depot_i_complete_with_tests]$ ruby -Itest test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
.....
Finished in 0.290143 seconds.

5 tests, 23 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 27300
```

Whose code are we testing when we check the validations?

- Is it ours?
  - Look at the products.rb file
  - All we wrote was the regular expression for image_url
  - We did not write the validates helper methods
- Whose business logic/requirements are we testing?
  - Ours
- Testing the shopping cart is where we will really test our own written code.

Unit tests

```
require File.dirname(__FILE__) + '/../test_helper'

class CartTest < Assertions::TestCase

end
```

- All the fixtures are loaded

Fixtures

```
ruby_book:
  title:       Programming Ruby
  description: Dummy description
  price:       1234
  image_url:   ruby.png

rails_book:
  title:       Agile Web Development with Rails
  description: Dummy description
  price:       2345
  image_url:   rails.png
```

Unit tests for the cart shopping cart

- Test adding unique products

```
test "test_add_unique_products" do
      cart = Cart.new
      rails_book = products(:rails_book)
      ruby_book = products(:ruby_book)
      cart.add_product rails_book
      cart.add_product ruby_book
      assert_equal 2, cart.items.size
      assert_equal rails_book.price + ruby_book.price,
            cart.total_price
end
```

- ruby –Itest test/unit/cart_test.rb

Unit tests for the cart shopping cart

- test adding duplicate products

```
def test_add_duplicate_product
      cart = Cart.new
      rails_book = products(:rails_book)
      cart.add_product rails_book
      cart.add_product rails_book
      assert_equal 2*rails_book.price, cart.total_price
      assert_equal 1, cart.items.size
      assert_equal 2, cart.items[0].quantity
end
```

- you need to know the internal details of the classes
  - not just that the class is actually exposing its details to the world
    - not very OO

Other assertions

message is optional

- assert(boolean, msg)
- assert_equal (expected, actual, msg)
  - assert_not_equal
- assert_nil (object, msg)
  - assert_not_nil
- assert_in_delta(expected_float,actual_float, delta, msg)
- flunk(message)
  - unconditional failure

Functional tests

- Testing the controller
  - they direct the show using incoming requests and sending out responses
  - they also direct the user from one page to the next
- In rails, code that tests a single controller is a functional test
- Functional tests are for things such as:
  - was the web request successful?
  - was the user redirected to the right page?
  - was the user successfully authenticated?
  - was the correct object stored in the response template?
  - was the appropriate message displayed to the user in the view?

An example functional test

- Consider the hypothetical 'posts' controller that lists 'user posts' on the home page

```
test "should get index" do
  get :index
  assert_response :success
  assert_not_nil assigns(:posts)
end
```

In the `test_should_get_index` test, Rails simulates a request on the action called `index`, making sure the request was successful and also ensuring that it assigns a valid `posts` instance variable.

Testing the product controller

- Let remind ourselves what the product controller does by firing up the application.
  - Go through all the functionality that we will test

Can also take parameters
get(:show, {'id' => "12"}, {'user_id' => 5})

- The first test - index

```
test "should get index" do
    get :index #you also have post, put, delete, head
    assert_response :success
    assert_not_nil assigns(:products)
end
```

:success - Status code was 200
:redirect - Status code was in the 300-399 range
:missing - Status code was 404
:error - Status code was in the 500-599 range

Checks if it is nil

Available request types for functional tests

If you're familiar with the HTTP protocol, you'll know that `get` is a type of request. There are 5 request types supported in Rails functional tests:

- `get`
- `post`
- `put`
- `head`
- `delete`

All of request types are methods that you can use, however, you'll probably end up using the first two more often than the others.

Data collections available after request

After a request has been made by using one of the 5 methods (`get`, `post`, etc.) and processed, you will have 4 Hash objects ready for use:

- `assigns` – Any objects that are stored as instance variables in actions for use in views.
- `cookies` – Any cookies that are set.
- `flash` – Any objects living in the flash.
- `session` – Any object living in session variables.

```
flash["gordon"]                flash[:gordon]
session["shmession"]           session[:shmession]
cookies["are_good_for_u"]      cookies[:are_good_for_u]

# Because you can't use assigns[:something] for historical reasons:
assigns["something"]           assigns(:something)
```

- The idea is to check the content of the hashes after the request has been made to ensure that the controller behaves as expected

Other variables available

- The ActionController::TestCase provides us with many useful variables and methods:
  - @controller – instance of this controller
  - @request – contains a web request object
  - @response – response object
- So the tests don't really need a webserver to be running to test the application – a virtual server is provided

Testing the product controller
A more complete test

```
test "should create product" do
    assert_difference('Product.count') do
      post :create, :product => {:title=>"New book",
                                 :description=>"hello",
                                 :price=>100,
                                 :image_url=>"ded.gif"}
    end

    assert_redirected_to product_path(assigns(:product))
  end
```

Defaults to a difference of '1'

Testing the product controller (cont.)

- All functional tests can be run by
  - rake test:functionals
  - More on testing?
    - An excellent resource is http://guides.rubyonrails.org/testing.html

Testing the views

- Testing the response of your application (views) by ensuring that the template contains specific HTML elements.
- Question - How would you test the response of the product controller index action?
- The assert_select tag allows you to do this.

assert_select

- Testing response:
  - assert_select "title", "Pragprog Books Store"
    - Testing the actual content

```
assert_select "div#cart" do
  assert_select "table" do
    assert_select "tr", :count=>3
    assert_select "tr.total-line td:last-of-type", "$57.70"
```

There should be three rows

Last column should be 57.70