6.170 Tutorial 8 - Rails Testing

Prerequisites

1. Basic understanding of how a Rails App works
2. Have completed Project 1 and 2

Goals of this tutorial

1. Understand why we need testing in developing applications
2. Know the distinctions between the three main types of tests (unit tests, functional tests, and integration tests) in Rails
3. Able to write basic tests for non-trivial model and controller methods.

Resources

- http://ofps.oreilly.com/titles/9780596521424/testing_id64728.html
- http://guides.rubyonrails.org/testing.html
- http://api.rubyonrails.org/

Why Test?

In the context of a small class project, writing automated tests probably seems like a waste of time. However, writing tests becomes extremely valuable once you start collaborating with other programmers on a live-deployed project.

- **Automated testing saves time**: yes, writing tests initially takes more time. However, if you plan on developing or maintaining your product over an extended period of time, the savings add up.
- **Automated testing improves reliability**: in website development, it is popular to deploy new features in short release cycles (2 weeks, 1 week, even daily). When using a continuous integration server for deployment, when bugs are detected, the deploy process is automatically stopped, preventing the user from being exposed to a buggy product. This is especially important with dynamic languages such as Ruby and Javascript because there is no static compilation step to catch syntax or type errors.
- **Automated testing improves collaboration:** It is difficult for developers to keep track of what their colleagues are doing. By committing regularly and running the automated tests, developers get instant feedback as to whether their commit broke somebody else's code.

It may be difficult to decide what to test. 100% code path coverage is simply a waste of time. Here are some common sense guidelines to get you started.

- **Write tests for all external APIs.** They may be publicly facing APIs, or internal APIs between different development teams (client team vs server team). Developers who are trying to integrate with your APIs can use the tests as documentation for that specific endpoint.
- **Write tests for the mission critical subset.** The mission critical subset depends on the business. For example, a high-capacity banking would have a large mission critical subset, whereas a blog would have a much smaller one (if any).
- **Refactor code to minimize tests, and learn from best practices.** How does one go about proving that one's code is not vulnerable to SQL injection attacks? It would be silly to try to write a test for every single possible input field on the website. Instead, we can use the following invariant: if all of your database access was through the ORM, then your code is safe from SQL injection attacks if and only if the ORM is safe from SQL injection attacks. Since you are fairly certain that ActiveRecord is safe, you are fairly certain that your application is safe. By refactoring common functionality into units (in this case, the ORM), you can drastically reduce the burden of testing.

For the purposes of 6.170, you are not expected to write nearly as many tests you might write in a continuously deployed, production environment. We are only interested that you can identify a few critical operations in your codebase (eg: external APIs), and write tests for those.

Automated tests are only a first line of defense. Don't waste too much time writing automated tests, and always remember to manually test the things that are difficult to automate. (eg, browser compatibility, user experience)

Testing in Rails

The default rails application comes with its own testing framework. It is located in the “test” folder:

```
/test
      /fixtures
      /functional
      /integration
      /unit
```

There are three types of tests in rails:

- **Unit:** test a single rails model
- **Functional:** test a single rails controller
- **Integration:** test the interaction between multiple rails controllers

What is the difference? Mostly it’s just convention to separate tests into multiple directories, but “Functional” and “Integration” tests have some additional functionality (eg, @request and @response) that is not accessible in the Unit tests.

Fixtures is a fancy word for sample data. Fixtures allow you to populate your testing database with predefined data before your tests run. Fixtures are database independent and assume a single format: YAML(a markup language well-suited for representing hierarchical data structure) .

Hello World

Here is how to write and run your first test:

```
>> rails new test_project
>> rails generate model hello
>> rake db:migrate
>> vim test/unit/hello_test.rb

      require 'test_helper'

      class HelloTest < ActiveSupport::TestCase
      test "the truth" do
        assert true, "hello, this test passed"
      end
      def test_the_lie
        assert_equal 1, 2, "hello, this test failed"
```

```
    end
    end

>> rake db:test:prepare
>> rake test
```

Lets go through this example line by line:

1. Creating a new model using “rails generate” will automatically make a corresponding test file in “test/unit/modelname_test.rb”
2. Each file has “require test_helper”. This imports a file located at “test/test_helper.rb”. This is especially useful if you want to define some custom helper functions for testing.
3. Tests are grouped into classes. The class is defined to be a subclass of “ActiveSupport::TestCase” which defines all of the “assert” statements. If you want to define custom “assert” statements, you can reopen the “ActiveSupport::TestCase” class in your “test_helper.rb”.
4. Each test is a function. Any function whose name begins with the characters “test” is assumed to be a test, and will be run when you type “rake test”. Notice that there are two ways to define a function. First, you can use the traditional “def function_name” syntax. However, ActiveSupport::TestCase defines a “test” function which takes a string name, replaces all the spaces with underscores, prepends “test_”, and performs an “class_eval” (ruby metaprogramming) to add the function to the class.
5. A test consists of normal ruby code + assert statements. A full list of available assert statements is in the Rails Guide.
6. `rake db:test:prepare` makes sure that the test database schema has the proper migrations. If you have pending migrations, this command will warn you appropriately.

Unit Tests

The previous example was simple enough to code on the fly.

This next example is located in the subdirectory called “unit_test_example”.

Lets first take a look at the models:

app/db/schema.rb

```
ActiveRecord::Schema.define(:version => 20121008184328) do

create_table "friendships", :force => true do |t|
  t.integer  "user_id"
```

```
  t.integer  "friend_id"
  t.datetime "created_at", :null => false
  t.datetime "updated_at", :null => false
end

create_table "users", :force => true do |t|
  t.string   "name"
  t.datetime "created_at", :null => false
  t.datetime "updated_at", :null => false
end

end
```

app/models/friendship.rb

```
class Friendship < ActiveRecord::Base
attr_accessible :user_id, :friend_id
belongs_to :user
belongs_to :friend, :class_name => 'User',
           :foreign_key =>'friend_id'
end
```

app/models/user.rb

```
class User < ActiveRecord::Base
attr_accessible :name
has_many :friends, :through => :friendships
has_many :friendships, :dependent => :destroy

def add_friend(other_user)
     ?????
end

def remove_friend(other_user)
     ?????
end

def friends_with?(other_user)
   ?????
end
end
```

There is a many-to-many relationship between Users, using the intermediate model “Friendship”. Hopefully this is familiar to you by now.

I decided to have two operations for my User model: *add_friend* and *remove_friend*, and a third function, *friends_with?,* to determine whether two people are friends. Unfortunately, at this point in time, I haven’t figured out how to implement the functions -- I only have a rough idea of what they are supposed to do.

Test Driven Development (TDD), is a philosophy where one writes tests to describe behavior before writing the code that implements that behavior. This way, the tests can be used to help debug your code as you write it.

Side Note: A “Test Driven” approach is used not only for software development, but also in other fields/industries like manufacturing (Toyota Production System) or even building startups (“Lean Startup”)

Lets try this out. Here, is the file that I wrote to describe the behavior of the User model.

test/unit/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  test "add and remove friends" do
      billy = users(:billy)
      johnny = users(:johnny)

      assert (not billy.friends_with?(johnny)),
           "billy should not be friends with johnny before"
      assert (not johnny.friends_with?(billy)),
           "johnny should not be friends with billy before"

      billy.add_friend(johnny)

      assert (billy.friends_with?(johnny)),
           "billy should be friends with johnny"
      assert (johnny.friends_with?(billy)),
           "johnny should be friends with billy"

      johnny.remove_friend(billy)

      assert (not billy.friends_with?(johnny)),
           "billy should not be friends with johnny after"
```

```
      assert (not johnny.friends_with?(billy)),
           "johnny should not be friends with billy after"
  end
end
```

Look at the first line of the test function:

```
     billy = users(:billy)
```

This somehow produces a User object named Billy. Where does it come from? In this case, we are using things called **fixtures**.

test/fixtures/user.yml

```
billy:
name: Billy

johnny:
name: Johnny
```

When the test suite is run, Rails does two things for you:

1. Resets the test database and loads any fixtures.
2. Makes the fixtures available in the tests using the above syntax

Fixture files are powerful. If necessary, you are allowed to use the “erb” templating language (the same one use for html files) to automatically generate fixture files. However, when the scope of your tests increase (such as when you do integration tests), modeling all of the relationships between models with fixture files can become tedious. If this becomes an issue, you should use the factory approach to pre-populating your test database, using a gem such as “FactoryGirl”.

Tip: it is a good idea to put international characters (such as “ó”) in your fixture data. It would be embarrassing for you web application to crash simply because someone’s last name has an accent mark.

Moving on: the rest of the test function should be straightforward. Currently, since none of the functions have been implemented, the assert statements should fail.

The completed version of this example is located in the repository.

Functional Tests

Functional tests are for individual controllers of your application. The code for this example is located in the subfolder called “functional_test_example”.

In this example, I am designing a webservice to help me add numbers together. It will behave roughly like this:

Request:

```
GET /add?num1=2&num2=4
```

Response:

```
200
Content-Type: text/plain
6
```

In the case that the input is invalid, it should return with a “400” Bad Request status code

Here are the tests describing the behavior of this webservice.

test/functional/math_controller_test.rb

```
class MathControllerTest < ActionController::TestCase
  test "add when input is correct" do
      get :add, { 'num1' => -2, 'num2' => 4}
      assert_response 200
      assert_equal "2", @response.body
      assert_equal "text/plain", @response.content_type
  end
  test "add should return 400 when params invalid" do
      get :add, {'num2' => 'blah'}
      assert_response 400
  end
end
```

Note that like the Unit test cases, functional test cases are subclasses of “ActionController::TestCase”. However, Rails magically gives you access to some request/response functions and variables:

Functions:

```
get
post
put
head
```

`delete`

Hashes:

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

Variables:

`@controller` – The controller processing the request (the object, not the class)
`@request` – The most recent request
`@response` – The most recent response

Here is the completed webservice:

app/controllers/math_controller.rb

```
class MathController < ApplicationController
  def add
      begin
          n1 = Integer(params[:num1])
          n2 = Integer(params[:num2])
          render :text => (n1+n2).to_s,
                :content_type => "text/plain"
      rescue
          render :status => 400, :text => "Invalid Input"
      end
  end
end
```

config/routes.rb

```
FunctionalTestExample::Application.routes.draw do
  match "add" => "math#add"
end
```

Integration Tests

Functional tests are good for testing single controller actions. However, many processes, such as the checkout process of a shopping website, do not occur on a single controller. In such cases, the user's experience spans across multiple controllers, and may be quite complicated. This is where integration tests become handy.

Integration tests, unlike Unit and Functional tests, are not generated automatically. To generate an integration test, you must type:

```
>> rails generate integration_test user_flow
```

For the sake of simplicity, in this example, I am only going to have one controller. However, the integration test will use multiple actions on that controller.

There will be two operations: *login* and *logout*

POST /login
    post-condition: session[logged_in] = True, redirect_to “/welcome”
POST /logout
    post-condition: session[logged_in] = False, redirect_to “/”

Note that this example will be extremely simple -- it won’t even check for passwords.

Here is the integration test:

test/integration/login_logout_test.rb

```
require 'test_helper'

class LoginLogoutTest < ActionDispatch::IntegrationTest
  test "login and then logout" do
      post_via_redirect "/login"
      assert_equal '/welcome', path
      assert_equal true, session[:logged_in]

      post_via_redirect "/logout"
      assert_equal '/', path
      assert_equal false, session[:logged_in]
  end
end
```

Integration tests have all of same helper methods as functional tests, plus a couple more relating to redirection and session management. For example, the above example uses the function `post_via_redirect` which performs a regular POST call, but then redirects to the subsequent page if applicable (just like a browser would). Additionally, there is an `open_session`, function which executes a block of code, and then returns a “browsing session” object, enabling you to perform integration tests involving multiple browsers. Take a look at the Rails Guides for more details

Here is the controller that completes the above test:

config/routes.rb

```
IntegrationTestExample::Application.routes.draw do
  root :to => "home#frontpage"

  match "login" => "sessions#login", :via => :post
  match "logout" => "sessions#logout", :via => :post
  match "welcome" => "home#welcomepage", :via => :get,
          :as => :welcomepage
  match "" => "home#frontpage", :via => :get, :as => :frontpage
end
```

app/controller/sessions_controller.rb

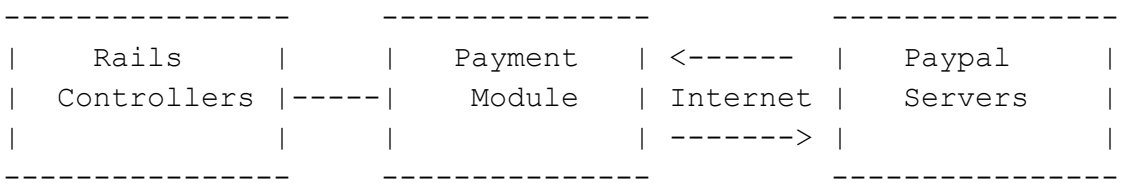
```
class SessionsController < ApplicationController
def login
    session[:logged_in] = true
    redirect_to welcomepage_path
end

def logout
    session[:logged_in] = false
    redirect_to frontpage_path
end
end
```

Stubbing and Mocking

Tests are supposed to be quick and self-contained. Unfortunately, the websites we create often have to communicate with external APIs (such as the Paypal) in order to function properly. How do we write self-contained tests if our code must communicate with 3rd party APIs?

This is where the power of dynamically typed languages comes into play. Suppose we just created a shopping website which does payment through paypal. If we had designed our code in a modular manner, we should end up with something like this:



As shown in the diagram, all communication with the Paypal Servers goes through our Payment Module. In a production environment, the payment module would communicate with the web to move money on paypal. However, in our development environment, we probably don't want to be making payments every single time we test our code. Additionally, communication with the Paypal server requires internet connectivity, causing tests to take a very long time, especially when one's internet connection is slow.

The solution? Metaprogramming. Before we run the test suite, we can substitute the real payment module with a fake payment module that doesn't actually communicate with the Paypal servers.

This can be done using pure Ruby, but there exist libraries that make it easier to swap out functions on the fly. One such library is **Mocha**. With Mocha, the syntax for replacing a function is:

```
PaymentModule.expects(:make_payment)\
      .with("100 dollars").returns(True)
```

The above code indicates the the PaymentModule is "supposed" to receive the *make_payment* function call with the specified arguments. If, during the course of the test, the *make_payment* function does not get called, then the test will throw an error.

Wrap Up

Like many things in the computer world, web application testing has generated a cult-like following in some circles. Just a quick glance at the testing section of the course textbook will give you a glimpse of the strange things that people have invented to test their web applications -- testings frameworks that read like plain english, weird exotic names --- cucumber, capybara, etc.

Especially during rapid development when you are experimenting with different ideas, TDD may actually be more of a hindrance than help. If TDD is not your style, just write a couple of tests for the most critical parts of your web application. On the other hand, if you are enthralled by the concept of integration tests that read like plain English, feel free to explore all of the weird testing frameworks in the Rails ecosystem. Testing is supposed to make the development process easier. If you don't feel that this is the case, you might want to reconsider your approach to testing.

Notes on Submitting Tests

- To facilitate grading, make sure to delete all tests that are generated by scaffolding and have not been modified by you.
- Make sure all tests have passed(eg. by running “rake test”) before turning in your submission.

MIT OpenCourseWare
http://ocw.mit.edu

6.170 Software Studio
Spring 2013

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.