Andrew Koelling

AJAX

So the book just kinda goes "lets make an ajax cart!" without really mentioning what AJAX does. In layman's terms, AJAX is a way for webpages to update their content without going through a full reload. An example of a world without AJAX would be like rebuilding an entire car because you need to change the tires. AJAX makes it so we don't need to rebuild the engine if were only changing the tires.

Due to the heavy use of javascript and the jquery module, most AJAX calls won't work if the browser has scripts disabled. To do the AJAX call normally, you would need to write quite a bit of javascript or use the jquery library, but rails abstracts this from us. So long as you update the application.html in views/layouts to include javascript on the application, the only thing you need to do to enable ajax calls is simply say :remote true on a button or link, and make a partial template for the content. Due to rails magic and other stuff, sometimes the prototype gem is required, just do gem install prototype.

Contrary to popular belief, AJAX still sends data to the server, AJAX is not a programming language but rather a way of using javascript to get particular results, and jquery is not required to use AJAX but the jquery abstracts away all the browser specific nonsense and provides a much easier to use interface. Rails will require jquery to do AJAX however due to its.

here is an example of a AJAX enabled button in rails. This would be in the view folder

```
<%= button_to 'Considering', line_pets_path(pet_id: pet), remote: true %>
```

Here is what you need to do to enable javascript in links on rails 4. this is in the application.html.erb file

```
<%= javascript_include_tag "application" %>
```

Then you make a partial for the updated content and thats how rails handles AJAX.

- http://www.tutorialspoint.com/ajax/what_is_ajax.htm
- http://en.wikipedia.org/wiki/Ajax_(programming)
- http://www.w3schools.com/ajax/ajax_intro.asp

Maria Deslis - Unit Testing in Rails

For this report, I decided to do a little more in depth research on unit testing since that was the part of the project I was focused on the most. Here are the main things I learned.

This is the basic skeleton of unit tests in ruby. Naming is very important, and when doing tests you should always name it after the subject you are testing with an underscore test (see below).

Unlike unit tests in Java, there is typically *one* assertion PER test, as opposed to a couple of assertions per test. The main idea for this is the resulting maintainability of the test, you can give a more descriptive name to your test. Tests that focus on one behavior of the system are almost always easier to write and to comprehend at a later date. If our class contained a bug in the initialize method, only the first failing assertion would be reported.

A good workflow when creating unit tests is:

1. Create your app skeleton/scaffold
    a. You can't write tests for something that doesn't exist yet
2. Write failing tests FIRST
    a. Before you start adding things to databases, changing layouts, adding forms, etc.
3. Write code to pass ONE TEST AT A TIME
    a. This will allow for easier debugging and faster productivity
4. Once your code passes all the tests, refactor your code
    a. Keep it simple and DRY

This is a workflow that I think should have done from the beginning of the semester and it would have allowed us to learn the code in a clearer, step by step way. *This means that from now when we start adding new parts to the code we need to make unit tests **first!***

Basic Skeleton of a unit test in rails. File : <subject>_test.rb

```
require "test/unit"

class <Subject>Test < Test::Unit::TestCase
        def test_<what_we_are_testing>
                <assertion>
         end
end
```

- http://courseware.codeschool.com.s3.amazonaws.com/rails_testing.pdf
- http://guides.rubyonrails.org/testing.html
- http://everydayrails.com/2011/01/11/beginning-rails-testing.html

---

Anastasia Shpurik - Test Fixtures

We had some difficulties with test fixtures, so I decided to research them some more. Fixtures are sample data that allow us to populate our test database with some predefined data to be used in our tests. They are written in YAML and are database independent. Rails creates fixtures for each model we create.

Each fixture is given a name and then an indented list of key/value pairs. When we run rake test, it takes these fixtures, loads them in the database and then tests our unit tests using this data. The issue that we were running into was for our foster_parent association, when a new foster_parent was created, the status of the pet was also updated. We were thus trying to access fixture data from pets.yml in our foster_parents.yml. The pet_id was dependent on the primary key of each pet. We thought that in the test database, the fixtures were loaded in incremental order for their primary keys. This was not the case, they are loaded as hashes, which kept causing our tests to fail. In general, having these autogenerated IDs is good for testing, but in the case of how our app was set up, we had to create our own IDs. To fix this, we added an id: field to pets.yml and manually populated these primary keys. Then, the foster_parent test was able to pass.

```
ruby:
  id: 3
  name: Ruby
  breed: Kitty
  image_url: ruby_kitty.gif
  age: 8.00
  habits: Cat Stuff
  status: Available
```

which allowed the following to pass:

```
@pet = Pet.find(foster_parent_params[:pet_id].to_i)
@pet.update_attribute(:status, "Fostered")
```

- http://guides.rubyonrails.org/testing.html#the-low-down-on-fixtures
- http://api.rubyonrails.org/v3.2.13/classes/ActiveRecord/Fixtures.html