INFO 2310

Topics in Web Programming
Ruby on Rails

Week 7

Pagination
&& AJAX
&& Avatars

Go ahead and login to Amazon and start your EC2 instances

Login to PuTTY

- open PuTTY
- On the left panel, navigate to Connection->SSH->Auth
  - Next to "Private key file for authentication:", click "Browse", and select the *.ppk file you created on the previous step.
- Then, navigate to Connection->Data
  - For "Auto-login username", type "ec2-user"
- Navigate to "Session" (the very top)
  - Copy the "Public Domain Name" of your EC2 instance; you can see this on the instances page of the Amazon console, when an instance is selected
  - Paste it into the "Host Name (or IP address)" field
- Click "Open"

Login to WinSCP

- Open WinSCP
  - Paste in your domain to "Host name", as you did in PuTTY
  - Type "ec2-user" for the "User name"
  - Click "..." to select your private key file
  - Click "Login"
- Set NotePad++ as the default editor.
  - Click Options->Preferences
  - Select "Editors" from the left tab
  - Click Add
  - Select "External Editor"
  - Find NotePad++ (C:\Program Files (x86)\Notepad++)
  - Click "Open", then "Okay"
  - Drag it to the top of the editor list

Today's branch

Since we are working on a new feature today, let's start on a feature branch

```
git status                          # should display nothing to commit
git checkout -b pagination          # checkout a new branch
git branch                          # view branches
```

lecture_7_spec.rb

Go ahead and download the lecture_7_spec.rb from

https://raw.github.com/goggin13/curails-mg343/master/spec/requests/lecture_7_spec.rb
(also linked from the course website syllabus page, under today's lecture).

Put the spec at **spec/requests/lecture_7_spec.rb**

Last week we used the 'faker' gem to generate a set of realistic dummy data. Now we have too much of a good thing, so we'll need to put some structure around paginating our users and micro_posts.

Pagination

Do you like writing pagination logic?

Me neither; just the type of boring task we don't want to rewrite ourselves.

Rails core will not help us in this case, but there's a widely used gem that will.

will_paginate to the rescue.

will_paginate

Let's add the **will_paginate** gem to our gem file

We'll also add **bootstrap_will_paginate** which will adjust the css around the pagination to play nicely with our existing bootstrap css.

gem 'faker'
gem 'will_paginate'
gem 'bootstrap-will_paginate'

Then run
bundle install

will_paginate

All it takes to start using will_paginate is to provide it with a collection built with the "paginate" method, which it has added to all of our ActiveRecord models.

e.g. in the controller

User.all

=>

@users = User.paginate(page: params[:page])

And in a view...

<%= will_paginate @users %>

Let's set it up together for our users index; then you can paginate a user's micro_posts on their profile page.

app/controllers/users_controller.rb

```
def index
      @users = User.paginate(page: params[:page])
...
```

app/views/users/index.html.erb

```
<%= will_paginate @users %>

<% @users.each do |user| %>
  <tr>
    <td><%= user.name %></td>
     ....
  </tr>
<% end %>

<%= will_paginate @users %>
```

bundle exec rspec spec/requests/lecture_6_spec.rb -e "paginating users"
should be passing if we got this stage right

Paginate User's MicroPosts

- In the **show** action of **app/controllers/users_controller.rb**, create a collection of micro_posts for that user, using the paginate method (hint; you need to create a new variable, unlike we did with @users previously).
- Instead of the default 30 per page, display just 10 micro_posts per page.
  - **@collection.paginate(page: params[:page], per_page: 10)**
- Update **app/views/users/show.html.erb** to use the **will_paginate** function with the collection you defined in the controller function

bundle exec rspec spec/requests/lecture_6_spec.rb -e "paginating micro_posts"

should be passing now

User's Feed Round I

We'll start with this function in **user.rb**

```
def feed(paginate_options={page: 1})
    micro_posts.paginate(paginate_options)
end
```

Currently this is just a paginated collection of the user's posts. Next week we will enhance it to include the user's posts AND everyone they are following.

User's Feed Round I

You are already **will_paginate** pro's, so we will just go through this together quickly.

Our goal is to render out the feed on the home page. So we'll start with creating a variable in the **home** controller action for our feed:

```
def home
  if signed_in?
    @feed = current_user.feed page: params[:page]
  end
end
```

User Feed Round I

Then on the home page, we can render them out:

```
<% if signed_in? %>
  <%= will_paginate @feed %>
  <div id="user_feed">
    <%= render @feed %>
  </div>
  <%= will_paginate @feed %>
<% else %>
  <p>
    <%= link_to 'Sign Up', new_user_path, class: 'btn btn-primary btn-large' %>
  </p>
<% end %>
```

bundle exec rspec spec/requests/lecture_7_spec.rb -e "user feed"
should tell us if we got it right

Posting from the home page

Let's update the home page so a user can post a MicroPost from there upon logging in (a la, Twitter, our soon to be forgotten rival).

We'll walk through these updates together.

Creating MicroPosts

Currently, when you visit **/micro_posts/new** you have to manually edit the user_id of the micro_post. This is silly.

Let's remove the "user_id" field from the micro_posts form (**app/views/micro_posts/_micro_post.html.erb**). altogether.

And then use the **build** method of a user's micro_posts collection to fix this.

current_user.micro_posts.build

This handy function "builds" a new micro_post that belongs to the current user. This micro_post is not saved to the database, but the user_id is already set = current_user.id.

```
def create
  @micro_post = current_user.micro_posts.build(params[:micro_post])
  ...
```

Now, whenever a micro_post is created, it will belong to the current_user.

To the home page

Now, we're set to drop our micro_post form into the home page.

```
<% if signed_in? %>

  <%= render 'micro_posts/form' %>

  <%= will_paginate @feed %>

  <div id="user_feed">
    <%= render @feed %>
  </div>
```

We'll also need to add an @micro_post variable to **app/controllers/static_pages_controller.rb** (which is no longer so aptly named...).

app/controllers/static_pages_controller.b

```
def home
  if signed_in?
    @micro_post = MicroPost.new
    @feed = current_user.feed page: params[:page]
  end
end
```

Page reloads?*

Let's take a look at how we can submit our new form with AJAX.

There are many existing solutions and opinions on the easiest, cleanest way to integrate AJAX with your Rails forms.

Here's a great blog entry on the many ways to AJAXify your Rails actions

-> http://blog.madebydna.com/all/code/2011/12/05/ajax-in-rails-3.html

Some primary features of Rails 4 (beta released last week!) are aimed at easing AJAX integration.

Check out the Rails 4 release notes here:

-> http://weblog.rubyonrails.org/2013/2/25/Rails-4-0-beta1/

*2008 called; they want their web page back.

Step 1 : Update the form

app/views/micro_posts/_form.html.erb

<%= form_for(@micro_post) do |f| %>

=>

<%= form_for(@micro_post, **remote: true**) do |f| %>

Content-Types

- html:
  - Server returns HTML, which can be inserted into the page
- json:
  - Server returns JSON objects, which we can manipulate via JavaScript as we please (perhaps creating DOM elements ourselves and inserting them into the page).
- **script:**
  - Server returns a snippet of JavaScript, which is evaluated.
  - This is the default when we set **remote: true**

Step 2 : Edit our controller

app/controllers/micro_posts_controller.rb

```
  def create
    @micro_post = current_user.micro_posts.build(params[:micro_post])

    respond_to do |format|
      if @micro_post.save
        ...
        format.js { render :partial => "micro_posts/show"  }
      else
        ...
        format.js { render :partial => "micro_posts/error"  }
      end
    end
  end
```

Step 3 : Create a _show.js partial

app/views/micro_posts/_show.js.erb

```
// Get the HTML for a rendered MicroPost
var html = '<%= j(render(:partial => "micro_post", :object => @micro_post))%>';

// create a JQuery DOM element we can insert (and hide it so we can fade it in)
var element = $(html).hide();

// Add it to the user feed, and then fade it in
$('#user_feed').prepend(element);
element.fadeIn(1000);

// Reset the post input to empty
$('#new_micro_post input[type="text"]').val('');
```

Step 4 : Create a _error.js partial

app/views/micro_posts/_errors.js.erb

```
var errors = $('<div class="alert alert-error"/>');

<% @micro_post.errors.full_messages.each do |error| %>
  errors.append('<p><%= j(error) %></p>');
<% end %>

// Display errors before the form
$('#new_micro_post').prepend(errors);
errors.fadeOut(3000);
```

script content-type

Let's walk through this one briefly; it was not intuitive for me, and reeks of Rails magic*.

User types a post and clicks "Create Micro post" button

*Rails Magic isn't a bad thing; it's just important to have an idea of what exactly Rails is doing for you.

These are the options we set on the form

<%= form_for(@micro_post, **remote: true**) do |f| %>

They tell Rails to intercept our form submission (which would reload the page, normally), package up the data the form *would* have sent, and send it to the server via AJAX.

Our controller **create** action receives the same data as it would have were the form submitted normally; but the **content-type** header tells it to return JavaScript instead of HTML.

Specifically, the content type maps the response to the line we added in our controller

format.js { render :partial => "micro_posts/show" }

The Rails server renders the 'micro_posts/show' partial in the same manner as if it were an HTML view; only in this case the embedded ruby snippets are producing JavaScript instead of HTML.

The Rails JavaScript that intercepted our form submission is waiting for the server to respond.

When the server responds (with the JavaScript code from our 'show' partial), the browser evaluates the JavaScript code that was returned.

This allows it to, among other things, insert elements into the DOM.

And clear the input field.

Thoughts

I like this solution because:

- We are reusing our server side ERB templates for displaying our MicroPosts. The logic for what a MicroPost looks like is still encapsulated inside the **_micro_post.html.erb** partial. This feels nice and DRY.
- Once I was over (what was for me) a conceptual hurdle of the server returning JS that was then executed by the browser, it feels pretty straightforward.

I dislike this solution because:

- Creating a MicroPost via AJAX is now coupled with the templates we wrote in **_micro_post.js.erb**.
- The JS we are writing in these partials doesn't feel very modular, and would prove trickier to reuse versus writing code in the normal **app/assets/javascripts** path.

Testing JavaScript

To test the JavaScript we just implemented, we need *something* to parse and run it.

A common solution for Rails is to use Selenium, which is an API (with bindings for Ruby along with almost every other popular language) for controlling a browser.

We do not have Selenium set up on the EC2 instances, and it would be a slightly tricky task since we would have to install and run a browser in 'headless' mode (EC2 instances don't have a monitor, in case you haven't noticed).

We'll punt on this here; but a definite must have for your future work in Rails.

this test only really checks the correct form elements are being rendered on the page

bundle exec rspec spec/requests/lecture_7_spec.rb -e "posting from the home page"

Avatars

Next week we will return to our User Feed, and spend the class implementing following/unfollowing other users and adding followed posts to our feed. In the remaining time today, let's set up avatars for our users.

Another useful gem:

paperclip

https://github.com/thoughtbot/paperclip

Paperclip abstracts away the messiness of file uploading, storage, and resizing. It's awesome.

ImageMagick

paperclip relies on a piece of software called **ImageMagick**, which performs image processing (thumbnailing, resizing, etc...)

To install it, we need to run the following command on our EC2 instance.

```
sudo yum install pecl make ImageMagick ImageMagick-devel gcc re2c
```

Add paperclip to our Gemfile

gem 'paperclip', '~> 3.0'

and install it

bundle install

Add the relevant fields to the user model

rails generate paperclip user avatar

bundle exec rake db:migrate
bundle exec rake db:test:prepare

config/environments/development.rb

Rails provides configuration files in the **config/** directory.

config/application.rb
provides default settings for your application.

These can be overridden on a per-environment basis with the files in
config/environments
e.g.
config/environments/development.rb
config/environments/production.rb
config/environments/test.rb

Add ImageMagick path

We need to tell Paperclip where we installed ImageMagick

The unix command

which convert (convert is one of the utilities that comes with ImageMagick)

tells us ImageMagick is installed at

/usr/bin

So we add this to **config/environments/development.rb**

Paperclip.options[:command_path] = "/usr/bin/"

Add the avatar file field to User model

```
class User < ActiveRecord::Base
  attr_accessor :password
  attr_accessible :email, :name, :password, :avatar
  has_attached_file :avatar,
                    :styles => {
                       :medium => "300x300>",
                       :thumb => "100x100>"
                    },
                    :default_url => "/images/:style/missing.png"
...
```

Add the fields

app/views/users/_form.html.erb

<%= f.file_field :avatar %>

app/views/user/show.html.erb

<%= image_tag @user.avatar.url(:medium) %>

app/views/user/index.html.erb

<%= image_tag user.avatar.url(:thumb) %>

Finally

download some default avatars

As a time saving measure, you can use the commands provided in lecture_7.txt to download the ones I selected.

If we've followed all the steps correctly, our paperclip specs should be passing:

bundle exec rspec spec/requests/lecture_6_spec.rb -e "paperclip"

Check it out!

Fire up the browser and upload some avatars

Commit time

```
git status                          # see what we modified
git add -A                          # add all the changes
git commit -m "authentication"


git checkout master                 # merge it back into master
git merge access_control

git push origin master              # github
git push heroku master              # heroku

# optional; run our Faker dummy data script against our Heroku database
heroku run rake db:reset
heroku run rake db:populate
```

Today we...

- Used the will_paginate gem to paginate our users and micro_posts
- Took a first swing at the user feed
- Allowed users to post from the home page via AJAX
- Added user avatars to our site