INFO 2310

Topics in Web Programming
Ruby on Rails

Some Motivation

http://www.inc.com/keith-cline/hiring-recruiting-hardest-jobs-to-fill-2013.html

"The 5 Hardest Jobs to Fill in 2013"

- Big Data
- Mobile
- Enterprise Software
- Cloud Computing
- **Ruby on Rails**

"When it comes to web development, it's difficult to find engineers who are qualified across all technologies (PHP, Python, Java, for instance), but Ruby on Rails talent is frequently the toughest to find. A lot of start-ups choose Ruby on Rails because of the frameworks it offers and the fast speed at which development can be done. "

Last Week on INFO2310

- Created the login form
- Used a cookie to remember who the current user is
- Customized our layout for the current user

Week 6

Signing out

&& Access Control

&& Gems (fake data, pagination && image uploads)

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 access_control      # checkout a new branch
git branch                          # view branches
```

Logging out

In our sessions_controller, implement the **destroy** function.

The destroy function should

- logout the current user (we already have the function to do this in our **SessionHelper** module; **sign_out_user**).
- set a flash notice on the page that reads "Logged out <email>"
- redirect to the home page.

Once these are passing, you got it
bundle exec rspec spec/requests/session_spec.rb -e "logging out"

Once those pass, all your tests should be passing :)
bundle exec rspec

lecture_6_spec.rb

Go ahead and download the lecture_6_spec.rb from

https://raw.github.com/goggin13/curails-mg343/master/spec/requests/lecture_6_spec.rb

(also linked from the course website syllabus page, under today's lecture).

Put the spec at **spec/requests/lecture_6_spec.rb**

I've forgone the more ideal method of organizing tests in a functional manner (grouping similar tests in the same file) in lieu of the more convenient method of delivering a lectures' worth of tests in a single file.

As we did last week with **session_spec.rb**, we will progress through this spec during today's lecture.

Access Control

Currently, there are no authorization levels enforced in our app. Anyone, logged in or not, can edit any content. Scary stuff.

Now that we know who the current user is, we can enforce the following:

I shouldn't be able to edit other people's accounts, or microposts.

I shouldn't be able to see the sign in page or sign up page when I'm already logged in.

before_filter

The common way to accomplish these type of actions in Rails is to use "before_filters".

You may use before_filters to hook into the request process, and run your code before certain controller actions.

They may be used to DRY up some common code between controller functions, or they can intercept a request and perform a redirect before it actually reaches the controller function.

We will use them to accomplish our goals from the previous slide.

before_filter

lets add this function to session_helper.rb

```
def redirect_home_if_signed_in
    redirect_to root_path if signed_in?
end
```

And update our actions for logging in and signing up

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  before_filter :redirect_home_if_signed_in, only: [:new, :create]
  ...
```

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_filter :redirect_home_if_signed_in, only: [:new, :create]
  ...
```

After this,
bundle exec rspec spec/requests/lecture_6_spec.rb -e "login form"
and
bundle exec rspec spec/requests/lecture_6_spec.rb -e "registration form"
should both be passing

Protect our users!!

We'll start with a private helper function in
app/controllers/users_controller.rb

```
private
    def redirect_unless_authorized
        @user = User.find(params[:id])
        # Write some code here that redirects home
        # and displays an error message "You are not authorized
        # to edit that user" if the current_user is not equal to @user
    end
```

Finish the commented function above; then use the "before_filter" command as we did in the previous slide to redirect users who try to ***:edit***, ***:update***, or ***:destroy*** users that aren't themselves

When you are done
bundle exec rspec spec/requests/lecture_6_spec.rb -e "user access control"
should be passing

DRY it up

As an added bonus of the previous code snippet, we can remove some duplicated code from our UserController functions.

Namely, the
@user = User.find(params[:id])
line from ***create***, ***destroy,*** and ***update***

Protect our MicroPosts!!

Perform the same task with our MicroPostsController:

- Create an analogous **redirect_unless_authorized** function at the bottom of the MicroPostsController; if the current user is not the user who created the MicroPost, redirect them to the home page using the error message "**You are not authorized to edit that MicroPost"**

- Set an appropriate before_filter so users may only ***:edit***, ***:update***, or ***:destroy*** micro_posts that they created.

- DRY up what we can of our MicroPostsController

bundle exec rspec spec/requests/lecture_6_spec.rb -e "micropost access control"

should be passing when you are done

UsersController#index

It would be nice for our users to all be able to see one another.

Add a "Users" link to our header that goes to the user index listing ("/users").

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

Should pass when you succeed

This is boring

Despite the inevitable notoriety/traffic our site will one day experience, we don't currently have very many users. Let's fix that.

We will utilize the "Faker" gem to create dummy data. This is a good tool for generating a bunch of realistic-ish data when you are developing locally.

Add faker to our gemfile

gem 'bootstrap-sass', '2.2.2.0'
gem 'faker'

Then install it

bundle install

Sample Data

In order to take advantage of Faker, we will need to create a "Rake task"

Rakefiles (whose name is a play on Makefiles from C) are used for automating common tasks.

We've been using them regularly;

bundle exec **rake** db:migrate

for example.

type "**bundle exec rake -D"** to see lots and lots more available

Let's see what one looks like to create some sample data

lib/tasks/sample_data.rake

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    User.create!(name: "Matt",
                 email: "goggin13@gmail.com",
                 password: "password")

    99.times do |n|
      user = User.create!(name: Faker::Name.name,
                          email: Faker::Internet.email,
                          password: "password")
      50.times do |i|
        user.micro_posts.create! content: "hello, world - #{i}"
      end
    end
  end
end
```

Other common rake use cases:

- clearing caches
- deploying
- sending daily emails

Let's setup our sample data

- bundle exec rake db:reset
- bundle exec rake db:populate
- bundle exec rake db:test:prepare

Check it out!

If you visit **/users,** you will see lots of users.

And if you look at a profile page, you will see lots of micro_posts.

Too many, I would say...

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

Avatars

Another useful gem:

paperclip
https://github.com/thoughtbot/paperclip

Which abstracts away the messiness of file uploading

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_6.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 "access control, avatars, and faker"
```

```
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...

- Implemented signing out
- Added access control to our users and micro_posts
- Used the **faker** gem and a custom **rake task** to generate some fake data
- Used the **will_paginate** gem to paginate our users and micro_posts
- Used the **paperclip** gem to let users upload avatars