ECE:5820/CS:5820, SELT
Fall 2018

Fourth Homework Assignment
Due Date: Monday, Sept. 30 by 11:59 p.m.

Important Note: This assignment involves adding functionality to the Rotten Potatoes application and deploying the enhanced app to a cloud-based hosting service called Heroku. **Before starting this assignment, you must carefully follow the instructions in the accompanying document, titled "Instructions for deploying rotten-potatoes to Heroku".** Don't wait until the last minute to push your assignment to Heroku for the first time. We will be grading this assignment by exercising your deployed app. If your app is not successfully deployed, we will not be able to grade it.

Introduction:

For this assignment you will add a simple login-based authentication mechanism to Rotten Potatoes. This mechanism will use a session cookie to keep the user logged in until she/he explicitly logs out or deletes the cookie from the browser. The user will be able to sign up for an account by supplying a unique user-ID and an e-mail address. For now, the user will not need to supply a password. Later in the semester, we will discuss management of secure passwords and other security issues related to authentication and session management.

Part 1: Add a User model to Rotten Potatoes:

Let's begin by adding a User model to Rotten Potatoes. You should first generate a new migration for the `users` database table. The `users` table should have the following attributes:

- `string - user_id`
- `string - email`
- `string - session_token`
- `timestamps – (anonymous)`

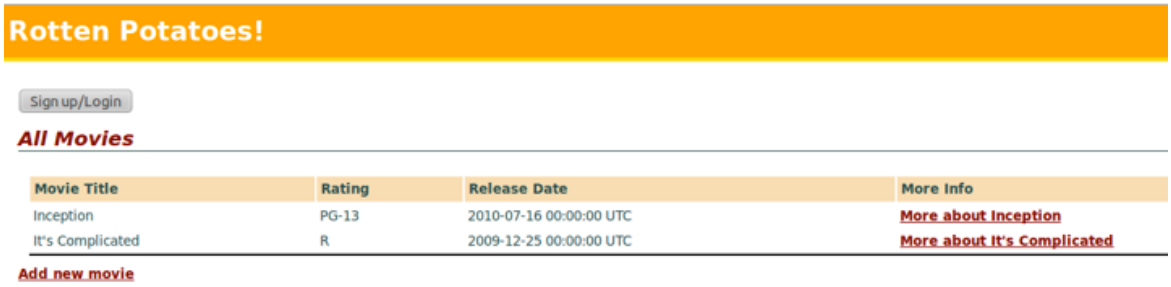
The 'timestamps' attribute is an internal mechanism, and it will not be associated with a symbol.
The attributes `user_id, email, and session_token` should all be strings. The `session_token` will be a randomly-generated string that will be used in later portions of the assignment. Using the `rake db:migrate` utility, apply the migration to the database.

Then define the `User` model class in the file `app/models/user.rb.` It will be empty for now, but we will add a method to it in the next part of the assignment.

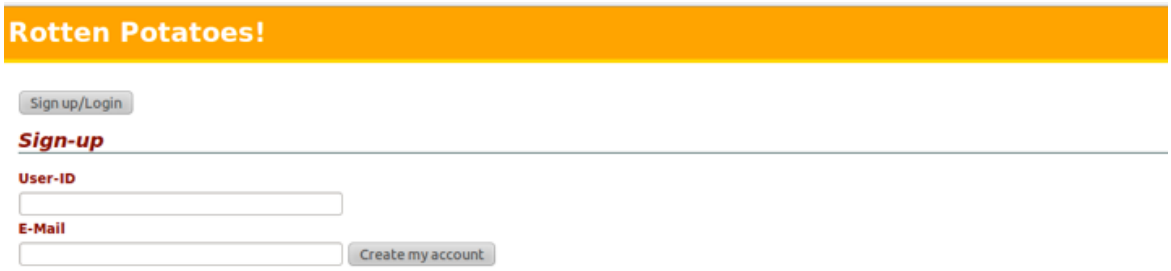
Part 2: Implement routes, controller and views for the `User` model:

Create the basic CRUD routes for your `UsersController` in `config/routes.rb`. Although we will use only two of the routes, it will not hurt to generate them all.

Next, we will add a button to allow users to sign up for an account. This button should be added to `app/views/layouts/application.html.haml` so that it is visible on all pages of the app. For now, link the button to the `new_user_path`. Use the rails `button_to` helper to create this button. When you are done, the Rotten Potatoes homepage should look like this:



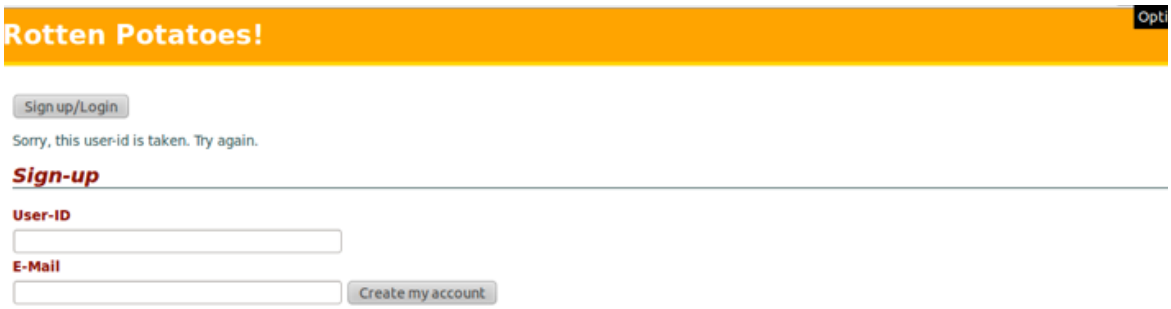
Next, create the `app/views/users` directory to hold the view associated with your `UsersController#new` action-i.e. new.html.haml . This view should display a form like the following:



You can refer `to app/views/movies/new.html.haml` for an example of creating a form view.

Now, you will need to implement the `UsersController` actions `new` and `create.` The `new` action simply renders the view: `app/views/users/new.html.haml`, while the `create` action must add the new user to the `User` model, using the data supplied by submission of the Sign-up form.

There are several important issues with the `UsersController#create` action. First, we want to insure that user-IDs are unique. Hence, before adding the new user to the database, the create action should check to see if an entry with this `user_id` already exists in the database. If so, the user should be directed to select a different user_id, as shown below: (Note: we do NOT require that e-mail addresses are unique.)



If the user account is successfully created, the user should be directed back to the Rotten Tomatoes home page, and a flash message of the form:
`'Welcome <user-ID>. Your account has been created'`
should be displayed.

A second issue with the `UsersController#create` action involves the `session_token` attribute. The `session_token` is a random string that will be used in the next part of the assignment to uniquely identify sessions. The `session_token` for a given user is assigned at the time the user is added the database. The token is generated by a call to the Ruby method:
`SecureRandom.base64`

In accordance with the philosophy of thin controllers, you should place the functionality to generate the `session_token` into the `User` model. In particular, you should define a method `User::create_user!` that takes as its parameter a hash with keys `:user_id` and `:email`. The method should add a third hash pair for the :`session_token` and then call the `ActiveRecord::create!` method to add the new user to the database. (Note that `create_user!` will be a class method of User.)
The `UsersController#create` method can now call this method, passing the `params` hash as an argument, to add a new user to the model.

To test the new functionality, fire up `rails server` and add several users via the web browser. Also, verify that your code properly checks for duplicate user-IDs. Now use `rails console` to look at the contents of the User model to verify that the users have been properly added to the database.

Part 3: Adding Login/Logout functionality and managing the session:

Let's begin by adding a `SessionsController` and associated routes for login/logout. This controller will be a little different than others that we have seen in that it is *not* associated with its own Model. The `SessionsController` will have three actions: `new`, `create` and `destroy`. Actions `new` and `create` will be associated with establishing a new login session while `destroy` will handle logout.

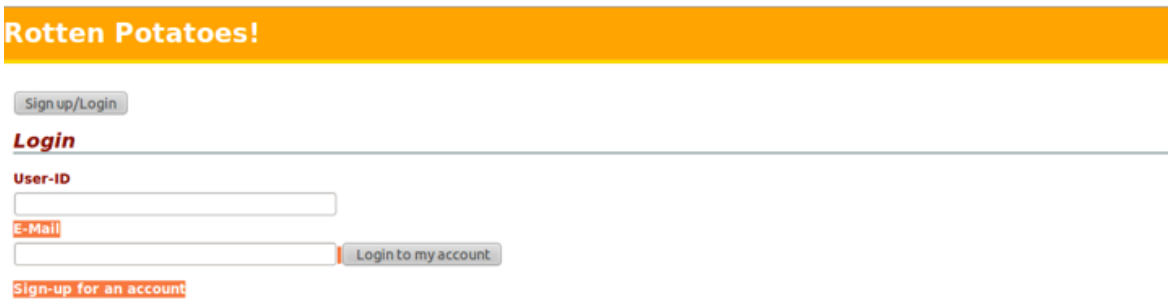
Since the nature of these actions is a little different from standard CRUD, we will define custom routes for them by adding the following entries to config/routes.rb:

```
match '/login', to: 'sessions#new', via: :get
match '/login_create', to: 'sessions#create', via: :post
match '/logout', to: 'sessions#destroy', via: :delete
```

Now run rake routes. You should see something like the following:

```
      movies GET    /movies(.:format)          movies#index
             POST   /movies(.:format)          movies#create
   new_movie GET    /movies/new(.:format)      movies#new
  edit_movie GET    /movies/:id/edit(.:format) movies#edit
       movie GET    /movies/:id(.:format)      movies#show
             PUT    /movies/:id(.:format)      movies#update
             DELETE /movies/:id(.:format)      movies#destroy
       users GET    /users(.:format)           users#index
             POST   /users(.:format)           users#create
    new_user GET    /users/new(.:format)       users#new
   edit_user GET    /users/:id/edit(.:format)  users#edit
        user GET    /users/:id(.:format)       users#show
             PUT    /users/:id(.:format)       users#update
             DELETE /users/:id(.:format)       users#destroy
       login GET    /login(.:format)           sessions#new
login_create POST   /login_create(.:format)    sessions#create
      logout DELETE /logout(.:format)          sessions#destroy
```

Now create a view for the `sessions#new` action in `app/views/sessions/new.html.haml` This view will display a login form (nearly identical to the Sign-up form) that looks like:



The "login to my account" button should link to the `login_create_path` and the "Sign-up for an account" link should link to `the new_user_path.`

Modify the UsersController#create action so that it redirects to the `login_path` rather than the `movies_path`.

Finally, change the button in `app/views/layouts/application.html.haml` so that it links to the `login_path.`

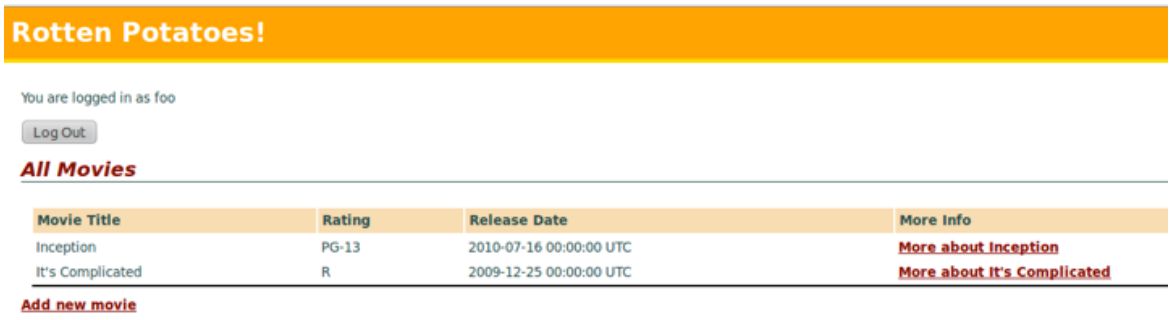
Note the overall effect of the above changes: When a user clicks the "Login/Sign-up" button she/he is taken to the Login page. If the user does not have an account, he/she can click the link on the Login page to be taken to the Sign-up page. Once the user has successfully signed up for an account he/she will be directed back to the Login page. After successful login, and after logout, the user is directed back to the Rotten Potatoes home page.

Now we are ready to implement the session management. We'll start by explaining the desired functionality. When a registered user signs in, the user will be identified by `user_id` in the `User` model. If the `user_id` is not found in the database or the supplied `email` address does not match the one in the database, the user should be redirected to the `login_path` with a flash message indicating an invalid user-id/e-mail combination.

If login is successful, the user's `session_token`, from the database, should be stored into the session cookie, as follows:

```
session[:session_token] = user.session_token
```

And the user should be redirected to the homepage (`movies_path`) with a message indicating successful login, as shown below:



Note that, in addition to the login message, the button has now changed from "Login/Sign-up" to "Logout". The message and "logout" button should continue to appear on all pages until the user logs out. When the user logs out (clicks the "Logout" button) the login message should disappear and the button should revert to "Login/Sign-up". Note that when the button displays "Login/Sign-up" it should be

linked to the `login_path`. When it shows "Logout" it should be linked to the `logout_path`.

So, how do we make this magic happen? Well, first we can note that, since we want the button and login message to appear on all pages, they must be dynamically generated in `app/views/layouts/application.html.haml`. Now, each controller action can set an instance variable `@current_user`, based on the value of the `session_token` in the session cookie, something like:

```
@current_user ||= session[:session_token ] &&
                  User.find_by_session_token(session[:session_token])
```

Then, `the application.html.haml` view can use this instance variable to generate the appropriate button and or message, in the following fashion (note that `@current_user` will be `nil` if the user is not logged in, since there will be no session token in the cookie)

```
-if !@current_user
    CODE HERE TO GENERATE SIGN_UP/LOGIN BUTTON
 -else
    CODE HERE TO GENERATE MESSAGE (using @current_user) AND LOGOUT
     BUTTON
```

However, there is a serious DRYness issue here since `@current_user` needs to be set by every `MoviesController` action. Fortunately, Rails provides a solution in the form of a mechanism called a before_filter. If the following line is placed at the beginning of the `MoviesController` class:

```
before_filter :set_current_user
```

it will cause the method `set_current_user` to be executed prior to executing any of the `MoviesController` actions. The `set_current_user` method can then be defined to set the `@current_user` instance variable as described above. Since the `set_current_user` method could conceivably be needed by other controllers, it is convenient to define this method in the parent class `ApplicationController` so that it will be inherited by all controllers. We will find additional uses for controller filters later in the semester.

Finally, we need to implement the logout functionality--i.e. the `SessionsController#destroy` action. This method should do the following:

- reset the session
- redirect to the homepage (`movies_path`)

Congratulations, you are done!! Of course, our simple login mechanism is not very secure since it doesn't utilize password protection. Later in the semester we will revisit the authentication process and introduce additional security measures. We will also look at some gems that help with login/logout and session management.

Important Additional Instructions:

It is essential that you follow these instructions EXACTLY or we will not be able to grade your submission:

1. The Sign up/Login button element must have id 'signupLogin'. Do NOT specify a :confirm option for this button.

2. The Log Out button element must have id 'logout'. Do NOT specify a :confirm option for this button.

3. For the Sign-up page:
   The User-ID field must have id 'signupUser'
   The E-Mail field must have id 'signupEmail'
   The "Create my account" button must have id 'signupCreate'

4. For the Login page:
   The User-ID field must have id 'loginUser'
   The E-Mail field must have id 'loginEmail'
   The "Login to my account" button must have id 'login_submit'
   The "Sign-up for an account" link must have id 'signup'

5. All flashes must have class 'flashMessage'

Below are two examples, illustrating how to add ids to HTML elements in haml

Example 1: Specifying 'add_title', 'add_rating', and 'add_submit' ids for elements on the new.html.haml page of Rotten Potatoes

new.html.haml

```
%h1 Create New Movie

= form_tag movies_path do

  = label :movie, :title, 'Title'
  = text_field :movie, 'title', :id => 'add_title'

  = label :movie, :rating, 'Rating'
  = select :movie, :rating, ['G','PG','PG-13','R','NC-17'], :id=> 'add_rating'

  = label :movie, :release_date, 'Released On'
  = date_select :movie, :release_date

  = submit_tag 'Save Changes', :id => 'add_submit'
```

Example 2: Adding 'delete' id for "Delete" button and 'to_edit' and 'back_to_movies' ids to respective links on the show.html.haml page of Rotten Potatoes. (Note: do NOT specify the :confirm option for your Signup/Login or Logout buttons.)

show.html.haml

```
-# in app/views/movies/show.html.haml

%h2 Details about #{@movie.title}

%ul#details
  %li
    Rating:
    = @movie.rating
  %li
    Released on:
    = @movie.release_date.strftime("%B %d, %Y")

%h3 Description:

%p#description= @movie.description

= link_to 'Edit', edit_movie_path(@movie), :id => 'to_edit'
= button_to 'Delete', movie_path(@movie), :method => :delete, :id => 'delete', :confirm => 'Are you sure?'
= link_to 'Back to movie list', movies_path, :id => 'back_to_movies'
```

Be sure to check the page sources for your app to double-check that all of your elements have the correct class/id. We will not regrade assignments that fail because of incorrect id/class attributes.

Submission Instructions (Be sure to follow these instructions *exactly!*):

Deploy your completed assignment to Heroku. We will be testing your deployed app on Heroku so it is e**ssential** that you successfully deploy your app to Heroku. The Heroku URI for your deployed app must be added to the GRADING.json file in your git repository **where**:

GRADING.json

```
{
  "heroku_url": "http://blah.blee.boo/myapp",
  "instructions": "Change the string associated with heroku_url to your
heroku application url to be graded."
}
```

Where the string http://blah.blee.boo/myapp is replaced with your URL.